Theory of Programming Languages

Hongseok Yang

December 11, 2023

Contents

Preface

Editor's Note:
This document is transcribed from Professor Yang's handwritten graduate programming language lecture notes for improved readability and accessibility.

Should you find any errors within these documents, contributions and error reports are welcomed at the following GitHub repository: github.com/p51lee/CS520-notes.

In this document, to express the additional arrowed notes of the original lecture notes as a footnote, I will be using symbols ♣, ♦, ♥, ♠, ★, ❀, and ✌ as footnote symbols.
So don't be surprised or confused if you see a footnote symbol in the middle of a sentence or an equation; just look down at the bottom of the page to see what it means. If you are using a PDF viewer, you can click on the footnote symbol to jump to the footnote.

Chapter 1

Predicate Logic

1.1 Motivation or objective

(1) Learn four key tools in PL that will be used throughout this course.

(i) Abstract Syntax

(ii) Denotational Semantics

(iii) Inference Rule

(iv) Binding

(2) Learn the basics of predicate logic (or first-order logic)

(3) We plan to go through some of (i) - (iv) twice, first using integer expressions and then using predicate logic.

1.2 Integer expressions

(1) How to analyze integer expression found in logic and programming languages mathematically? We will first have to define the syntax and the semantics for them.

(2) Examples: $x + 3 \times y$, $x \div 2 + x \times x$ ...

(3) We also want to develop mathematical tools to reason about or manipulate integer expressions.

1.3 Abstract syntax and initial algebra

(1) Abstract Syntax:
Specification of *abstract phrases*[♣] in a formal language, such as the language of integer expressions and predicate logic.

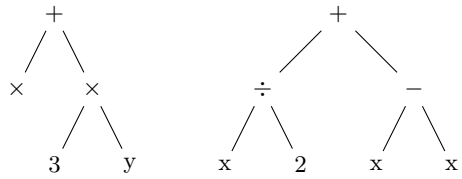
[♣]vague words, but will be made rigorous when we define initial algebra.

(2) Typically, we use *abstract grammar*♣ to describe abstract syntax.

(3) Abstract grammar for integer expressions:

$$\begin{aligned}\langle intexp\rangle \ ::= \ & 0 \mid 1 \mid 2 \ ... \\ \mid \ & \langle var\rangle \\ \mid \ & \text{-} \ \langle intexp\rangle \\ \mid \ & \langle intexp\rangle + \langle intexp\rangle \\ \mid \ & \langle intexp\rangle - \langle intexp\rangle \\ \mid \ & \langle intexp\rangle \times \langle intexp\rangle \\ \mid \ & \langle intexp\rangle \div \langle intexp\rangle\end{aligned}$$

(abstract) integer expressions are finite derivation trees in this grammar. For instance,



Note that infinite trees are not included.

(4) A more accurate view is to view abstract syntax as an initial algebra. This view will help us to see shy we can define various operations on abstract phrases or integer expressions using syntax-directed definition.

(5) *Algebra* $A \cdots$ Set with operations and constraints.
Signature $S \cdots$ Type of an algebra.

Example.

$$S_{\text{group}} = \left(t,\ \text{id} : t,\ \text{o} : t \times t \to t,\ (-)^{-1} : t \to t\right)$$

$$\begin{aligned} A_0 &= (\mathbb{Z},\ 0,\ +,\ -) \\ A_1 &= \left(\mathbb{R}_{>0},\ 1,\ \times,\ (-)^{-1}\right)\end{aligned}$$

$A_0 : S_{\text{group}}$ is a group with integers and addition. $A_1 : S_{\text{group}}$ is a group with positive reals and multiplication.

♣Here is the explanation of the word with an enumerated list:
(i) grammar without any concern on parsing for surface syntax.
(ii) In this case, parse trees in the grammar are abstract phrases.

(6) *Algebra homomorphism* $\cdots$ map between algebras that preserves constants and operations.

$$S = (t,\ c_1 : t,\ \cdots,\ c_n : t,\ \mathrm{op}_1 : t \times \cdots \times t \to t,\ \cdots \mathrm{op}_m : t \times \cdots \times t \to t)$$
$$A_0 = \left(\mathcal{U}_0{}^{\clubsuit},\ c_1^0 \in \mathcal{U},\ \cdots,\ c_n^0 \in \mathcal{U},\ \mathrm{op}_1^0 : \mathcal{U} \times \cdots \times \mathcal{U} \to \mathcal{U},\ \cdots \mathrm{op}_m^0 : \mathcal{U} \times \cdots \times \mathcal{U} \to \mathcal{U}\right)$$
$$A_1 = \left(\mathcal{U}_1,\ c_1^1 \in \mathcal{U},\ \cdots,\ c_n^1 \in \mathcal{U},\ \mathrm{op}_1^1 : \mathcal{U} \times \cdots \times \mathcal{U} \to \mathcal{U},\ \cdots \mathrm{op}_m^1 : \mathcal{U} \times \cdots \times \mathcal{U} \to \mathcal{U}\right)$$

$f \in \mathcal{U}_0 \to \mathcal{U}_1$ is a *homomorphism* if

(i) $f\left(c_i^0\right) = c_i^1$ for all i.

(ii) $f\left(\mathrm{op}_i^0\left(x_1,\ \cdots,\ x_k\right)\right) = \mathrm{op}_i^1\left(x_1,\ \cdots,\ x_k\right)$ for all i.

(7) *Initial algebra of a signature* S

(i) An algebra A of the signature S s.t. for all algebras A' of the same signature, there is a *unique* homomorphism f from A to A'.

(ii) A_{grammar} is initial.

(iii) Formally, an abstract syntax fixes a signature and it denotes an initial algebra of the signature. An abstract phrase is an element of that algebra.

Exercise 1.1. Prove that A_{grammar} is indeed an initial algebra.

Exercise 1.2.

Let A_0 and A_0 be initial algebras of the same signature S. Then, there are homomorphisms $f \in |A_0| \to |A_1|$ and $g \in |A_1| \to |A_0|$ s.t. $f \circ g = \mathrm{id}$ and $g \circ f = \mathrm{id}$. This means that all initial algebras of S are essentially the same, i.e. isomorphic. Prove this fact.

1.4 Syntax-directed definition and denotational semantics

(1) Definition of a map on integer expressions using a form of induction and case analysis.

(2) FV (Free Variables) : $\langle intexp \rangle \to 2^{\langle Var \rangle}$

$$\mathrm{FV}\left(e^{\blacklozenge}\right) = V^{\blacktriangledown}$$
$$\mathrm{FV}\left(c^{\spadesuit}\right) = \phi$$
$$\mathrm{FV}\left(x^{\bigstar}\right) = \{x\}$$
$$\mathrm{FV}\left(-e\right) = \mathrm{FV}\left(e\right)$$
$$\mathrm{FV}\left(e_1 \begin{matrix} + \\ - \\ \times \\ \div \end{matrix} e_2\right) = \mathrm{FV}\left(e_1\right) \cup \mathrm{FV}\left(e_2\right)$$

$^{\clubsuit}$notation: $|A_0|$
$^{\blacklozenge}$integer expression
$^{\blacktriangledown}$Set of free variables in e
$^{\spadesuit}$constant
$^{\bigstar}$variable

(3) Two features: case analysis, recursive calls on subphrases.

(4) $[\![-]\!] \in \langle intexp \rangle \to \Sigma \to \mathbb{Z}$ where $\Sigma = \langle var \rangle \to \mathbb{Z}$, a set of states σ.

$$
\begin{aligned}
[\![c]\!]\,\sigma &= c \\
[\![x]\!]\,\sigma &= \sigma\,(x) \\
[\![-e]\!]\,\sigma &= -\left([\![e]\!]\,\sigma\right) \\
\left[\!\left[e_1 \begin{array}{c} + \\ - \\ \times \\ \div \end{array} e_2 \right]\!\right]\sigma &= \left([\![e_1]\!]\,\sigma\right) \begin{array}{c} + \\ - \\ \times \\ \div \end{array} \left([\![e_1]\!]\,\sigma\right) \clubsuit
\end{aligned}
$$

Intuitively, $[\![-]\!]$ maps tress to mathematical functions in a syntax-directed (also called compositional) way. Such a compositional mapping from syntactic entities to mathematical entities is called *denotational semantics*.

(5) In both cases, we are using the initiality of $\langle intexp \rangle$. What are the target algebras in those cases?

(i)

$$
\begin{aligned}
|A_1| &= 2^{\langle var \rangle} \\
\underline{c^1} &= \phi \\
\underline{x^1} &= \{x\} \\
\underline{-^1}\,(X) &= X \\
\underline{+^1}\,(X, Y) &= X \cup Y
\end{aligned}
$$

$\underline{\times^1}$, $\underline{\div^1}$ and $\underline{\mathrm{rem}^1}$ are defined similarly.

(ii)

$$
\begin{aligned}
|A_2| &= \Sigma \to \mathbb{Z} \\
\underline{c^2}\,(\sigma) &= c \\
\underline{x^2}\,(\sigma) &= \sigma\,(x) \\
\underline{-^2}\,(f)\,(\sigma) &= -f\,(\sigma) \\
\underline{+^2}\,(f, g)\,(\sigma) &= f\,(\sigma) + g\,(\sigma)
\end{aligned}
$$

$\underline{\times^2}$, $\underline{\div^2}$ and $\underline{\mathrm{rem}^2}$ are defined similarly.

Exercise 1.3.

$$\delta \in \langle var \rangle \to \langle intexp \rangle \quad \text{(substitution)}$$

Define e/δ, the application of substitution δ to e.

$\clubsuit$some special treatment of the divide-by-zero case

1.5 Structural induction

(1) We want to show that some property Ψ holds for all integer expressions. What should we do?

(2) Use induction on the structure of expressions. That is, for each expression e, prove the property assuming that the property holds for the subexpressions of e.

Lemma 1.1 (Coincidence)**.** For every expression e and states σ and σ', if $\sigma(x) = \sigma'(x)$ for all $x \in \mathrm{FV}(e)$, then $[\![e]\!]\sigma = [\![e]\!]\sigma'$.

Proof. By structural induction.

- $e \equiv c$: $[\![c]\!]\sigma = c = [\![c]\!]\sigma'$.
- $e \equiv x$: $[\![x]\!]\sigma = \sigma(x) = \sigma'(x) = [\![x]\!]\sigma' \because x \in \mathrm{FV}(x)$.
- $e \equiv -e'$: $[\![-e']\!]\sigma = -[\![e']\!]\sigma = -[\![e']\!]\sigma' = [\![-e']\!]\sigma' \because$ induction hypothesis.
- $e \equiv e_1 + e_2$: $[\![e_1 + e_2]\!]\sigma = [\![e_1]\!]\sigma + [\![e_2]\!]\sigma = [\![e_1]\!]\sigma' + [\![e_2]\!]\sigma' = [\![e_1 + e_2]\!]\sigma'$.
- $e_1 \underset{\div}{\overset{-}{\times}} e_2$: similar.

□

Lemma 1.2 (Substitution)**.**

$$\sigma(x) = [\![\delta(x)]\!]\sigma' \quad \text{for all } x \quad \Longrightarrow \quad [\![e/\delta]\!]\sigma' = [\![e]\!]\sigma$$

Proof. By structural induction. □

Notation:

(i) $x_1 \to e_1, x_2 \to e_2, \cdots, x_n \to e_n$ means the substitution that maps x_i to e_i and $y \neq x_i$ to y.

(ii) $[\sigma \mid x : v](y) = \begin{cases} \sigma(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases}$

Corollary.

$$[\![e/x_1 \to e_1, \cdots, x_n \to e_n]\!]\sigma = [\![e]\!]\left[\sigma \;\middle|\; x_1 : [\![e_1]\!]\sigma \middle| \cdots \middle| x_n : [\![e_n]\!]\sigma\right]$$

This intuitively says the correspondence between syntactic and semantic substitutions.

(3) Structural induction holds because of the initiality of $\langle intexp \rangle$. Can you explain why it is the case?

1.6 Predicate logic (or first-order logic) informally

(1) Language for expressing (boolean) properties (also called assertions).

(2) Extension of boolean expressions in programming languages with universal and existential quantification.

(3) Examples:
$\forall x.\ \forall y.\ \exists m.\ \exists n.\ x \times m + y \times n = 1$
$\forall x.\ \exists y.\ y > x$

(4) Quantifiers are over integers, reals, and other first-order entities (i.e. not over sets of integers, and functions etc.). The "first-order" in first-order logic refers to this restriction. We will consider a version of predicate logic or first-order logic where quantifiers range over integers and all variables are integer variables.

1.7 Abstract syntax of predicate logic

(1) Described in terms of the following abstract grammar:

$$\langle intexp \rangle ::= 0 \mid 1 \mid 2 \mid ... \mid \langle var \rangle^{\clubsuit} \mid \text{-}\ \langle intexp \rangle \mid \langle intexp \rangle \overset{+}{\times} \langle intexp \rangle$$

$$\begin{aligned} \langle assert \rangle ::= &\ \text{true} \mid \text{false} \mid \langle intexp \rangle \overset{=}{>} \langle intexp \rangle \\ \mid &\ \neg\ \langle assert \rangle \mid \langle assert \rangle \wedge \langle assert \rangle \mid \forall\ \langle var \rangle\ .\ \langle assert \rangle \end{aligned}$$

To simplify presentations, we will consider only $+$, $-$, $\times$, $=$, $>$, $\neg$, $\wedge$, and $\forall$.

(2) What do we mean by abstract grammar and abstract syntax here? If you got confused about initial-algebra stuff, just think that our abstract syntax is the set of all finite derivation or parse trees. If not you can view the abstract syntax as the initial algebra of the signature induced by the grammar signature S_{PL}:

$$\text{Signature} \quad S_{\mathrm{PL}} = \left(\begin{array}{c} t^{\blacklozenge},\ u^{\blacktriangledown},\ 0 : t,\ 1 : t\ , 2 : t,\ \cdots, \\ x : t,\ y : t,\ \cdots, \\ - : t \to t,\ + : t \times t \to t,\ \times : t \times t \to t, \\ \text{true} : u,\ \text{false} : u,\ =: t \times t \to u,\ >: t \times t \to u, \\ \neg : u \to u,\ \wedge : u \times u \to u,\ \forall : \langle var \rangle \times u \to u \end{array} \right)$$

$$\text{Algebra} \quad A_0 = \left(\begin{array}{c} \mathcal{U}^0,\ \mathcal{V}^0,\ 0^0 \in \mathcal{U}^0,\ 1^0 \in \mathcal{U}^0,\ 2^0 \in \mathcal{U}^0,\ \cdots, \\ x^0 \in \mathcal{U}^0,\ y^0 \in \mathcal{U}^0,\ \cdots, \\ - \in \left[\mathcal{U}^0 \to \mathcal{U}^0 \right],\ \overset{+}{\times} \in \left[\mathcal{U}^0 \times \mathcal{U}^0 \to \mathcal{U}^0 \right], \\ \text{true}^0 \in \mathcal{V}^0,\ \text{false}^0 \in \mathcal{V}^0,\ \overset{=^0}{<^0} \in \left[\mathcal{U}^0 \times \mathcal{U}^0 \to \mathcal{V}^0 \right], \\ \neg^0 \in \left[\mathcal{V}^0 \to \mathcal{V}^0 \right],\ \wedge^0 \in \left[\mathcal{V}^0 \times \mathcal{V}^0 \to \mathcal{V}^0 \right], \\ \forall^0 \in \left[\langle var \rangle \times \mathcal{V}^0 \to \mathcal{V}^0 \right] \end{array} \right)$$

$^{\clubsuit}$some countably-infinite set disjoint from 0, 1, 2, ...
$^{\blacklozenge}$integer expression
$^{\blacktriangledown}$assertions

$$\text{Algebra} \quad A_1 = \begin{pmatrix} \mathcal{U}^1,\ \mathcal{V}^1,\ 0^1 \in \mathcal{U}^1,\ 1^1 \in \mathcal{U}^1,\ 2^1 \in \mathcal{U}^1, \\ \cdots \\ \forall^1 \in \left[\, \langle var \rangle \times \mathcal{V}^1 \to \mathcal{V}^1 \,\right] \end{pmatrix}$$

An algebra homomorphism from A_0 to A_1 is a pair $h : \mathcal{U}^0 \to \mathcal{U}^1$ and $k : \mathcal{V}^0 \to \mathcal{V}^1$ that preserves all constraints and operations. For instance,

(i) $k\left(=^0 (a,b)\right) \quad = \quad =^1 \left(h(a), h(b)\right)$

(ii) for any $\begin{smallmatrix} x \in \langle var \rangle \\ a \in \mathcal{U}^0 \end{smallmatrix}$, $k\left(\forall^0 \left(x, =^0 (a,x)\right)\right) \quad = \quad \forall^1 \left(x, =^1 \left(h(a), h(x)\right)\right)$ ♣

As before, the abstract syntax is the initial algebra of this signature S_{PL}.♦ Because of initiality, we can define a function in a syntax-directed way. Also, we can use structural induction to prove properties of assertions.

1.8 Denotational semantics

(1) We define two functions:

$$[\![-]\!]_{\text{intexp}} \in \left[\, \langle intexp \rangle \to \Sigma^{\heartsuit} \to \mathbb{Z} \,\right]$$
$$[\![-]\!]_{\text{assert}} \in \left[\, \langle assert \rangle \to \Sigma \to \mathbb{B}^{\spadesuit} \,\right]$$

(2) the definition of $[\![-]\!]_{\text{intexp}}$ is the same as before. Recall that it is syntax-directed.

(3) The definition of $[\![-]\!]_{\text{assert}}$ is as follows:

$$\begin{aligned}
[\![\text{true}]\!]_{\text{assert}}\, \sigma &= \mathbf{tt} \\
[\![\text{false}]\!]_{\text{assert}}\, \sigma &= \mathbf{ff} \\
[\![e_1 \overset{=}{>} e_2]\!]_{\text{assert}}\, \sigma &= \left([\![e_1]\!]\, \sigma \overset{=}{>} [\![e_2]\!]\, \sigma\right) \\
[\![\neg p]\!]_{\text{assert}}\, \sigma &= \left(\neg\, [\![p]\!]_{\text{assert}}\, \sigma\right) \\
[\![p_1 \wedge p_2]\!]_{\text{assert}}\, \sigma &= \left([\![p_1]\!]_{\text{assert}}\, \sigma \wedge [\![p_2]\!]_{\text{assert}}\, \sigma\right) \\
[\![\forall x.\ p]\!]_{\text{assert}}\, \sigma &= \left(\forall n \in \mathbb{Z}.\ [\![p]\!]_{\text{assert}}\, ([\sigma \mid x : n])\right)
\end{aligned}$$

(4) Don't forget that what appears inside $[\![]\!]$ is a tree, while $\forall$ and $\wedge$ on the RHS of $=$ are the usual mathematical notations. As we discussed already, here we are really defining an algebra A of S_{PL} s.t.

$$A = (\Sigma \to \mathbb{Z}, \Sigma \to \mathbb{B}, \cdots)$$

Then, we are using the initiality of the abstract syntax to get a map from it to A.

♣ same x

♦ isomorphic to the algebra built with derivation trees

♥ $\langle var \rangle \to \mathbb{Z}$

♠ $\{\mathbf{tt}, \mathbf{ff}\}$

1.9 Inference rules

(1) Rules for deriving always-true (in other words, valid[♣]) assertions.

(2) Expressed using the inference-rule notation.

$$\frac{p_0 \qquad p_0 \to p_1}{p_1} \qquad \frac{p}{\forall x.\ p} \qquad \frac{}{e_1 = e_2 \to e_1 = e_2}$$

general form: $\dfrac{p_0 \qquad \cdots \qquad p_n}{p}$

(i) expresses if all of $p_0, \cdots, p_n$ are valid, then p is valid.

(ii) doesn't say that for all $\sigma \in \Sigma$, if all of $[\![p_0]\!]_{\text{assert}}\,\sigma, \cdots, [\![p_n]\!]_{\text{assert}}\,\sigma$ are $\mathbf{tt}$, then $[\![p]\!]_{\text{assert}}\,\sigma$ is $\mathbf{tt}$[♦].

Exercise 1.4. Prove why the above three rules are correct[♥].

(3) A big part of research on or study about predicate logic is to study these rules. In this course, however, we will not do this.

1.10 Binding and substitution

(1) $\forall v$, $\exists v$:

(i) example of binders

(ii) They have the scope of binding

(iii) Informally, they introduce a new variable, give a nave v to it, and make it available within its scope. Morally, renaming v to some other variable w should not change the meaning of the assertion.

(2) An occurrence of a variable x is bound in p if x is within the scope of a binder for x in p.

(3) An occurrence of x is free in p if it is not bound.

(4) A variable is free in p if it has a free occurrence in p.

[♣] p is valid if $[\![p]\!]_{\text{assert}} = \mathbf{tt}$ for all $\sigma \in \Sigma$

[♦] σ satisfies p or p holds for σ.

[♥] also called sound.

(5) We can define functions $\mathrm{FV}_{\mathrm{assert}}$ and $\mathrm{FV}_{\mathrm{intexp}}$ ♣ that compute the sets of free variables of assertions and integer expressions, in a syntax-directed way:

$$\begin{aligned}
\mathrm{FV}_{\mathrm{intexp}}(e) &\cdots \text{defined as above} \\
\mathrm{FV}_{\mathrm{assert}}(\mathrm{true}) &= \phi \\
\mathrm{FV}_{\mathrm{assert}}(\mathrm{false}) &= \phi \\
\mathrm{FV}_{\mathrm{assert}}(e_1 \overset{=}{>} e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}_{\mathrm{assert}}(\neg p) &= \mathrm{FV}(p) \\
\mathrm{FV}_{\mathrm{assert}}(p_1 \wedge p_2) &= \mathrm{FV}(p_1) \cup \mathrm{FV}(p_2) \\
\mathrm{FV}_{\mathrm{assert}}(\forall x.\, p) &= \mathrm{FV}(p) \setminus \{x\}
\end{aligned}$$

Exercise 1.5.

Define an algebra for S_{PL} such that the algebra homomorphism from the abstract syntax to this algebra is $(\mathrm{FV}_{\mathrm{intexp}}, \mathrm{FV}_{\mathrm{assert}})$.

(6) Now, how should we deal with binders during substitution? It is not entirely obvious. Several textbooks had wrong definitions in old days.♦
Correct definition:

$$\mathrm{true}/\delta = \mathrm{true} \qquad \mathrm{false}/\delta = \mathrm{false}$$

$$\left(e_1 \overset{=}{>} e_2\ /\delta\right) = \left(e_1/\delta \overset{=}{>} e_2/\delta\right) \qquad \neg p/\delta = \neg\left(p/\delta\right)$$

$$(p_1 \wedge p_2)/\delta = \left(p_1/\delta \wedge p_2/\delta\right) \qquad (\forall x.\, p)/\delta = \forall v_{\mathrm{new}}.\, p/[\delta \mid v : v_{\mathrm{new}}]$$

$$\text{where} \quad v_{\mathrm{new}} \notin \bigcup_{w \in \mathrm{FV}(p) \setminus \{v\}} \mathrm{FV}\left(\delta(w)\right) \text{♥}$$

Property 1.1 (Coincidence)**.**

$$[\![p]\!]\,\sigma = [\![p]\!]\,\sigma'$$

$p \cdots$ assertion or integer expression
$\sigma, \sigma' \cdots$ states s.t. $\sigma(w) = \sigma'(w)$ for all $w \in \mathrm{FV}(p)$.

Proof. By structural induction.♠

- $p \equiv$ true or false: trivial.
- $p \equiv e_1 \overset{=}{>} e_2$:

$$\mathrm{FV}(e_i) \subseteq \mathrm{FV}(p) \quad \text{for } i = 1, 2 \quad \therefore \forall w \in \mathrm{FV}(e_i),\ \sigma(w) = \sigma'(w)$$

♣ subscripts to be omitted
♦ mistake: $\forall y.\ y > x/x \to y = \forall y.\ y > y + 1$
♥ device that lets us avoid variable capturing; if v is not in the set then $v_{\mathrm{new}} = v$. Otherwise, v_{new} is the first variable not in the set.
♠ we can use it because the abstract syntax for predicate logic is an initial algebra.

We can use induction and get $[\![e_i]\!]\,\sigma = [\![e_i]\!]\,\sigma'$.

$$\begin{aligned}[\![p]\!]\,\sigma &= [\![e_1]\!]\,\sigma \overset{=}{>} [\![e_2]\!]\,\sigma \\ &= [\![e_1]\!]\,\sigma' \overset{=}{>} [\![e_2]\!]\,\sigma' \\ &= [\![p]\!]\,\sigma'\end{aligned}$$

- $p \equiv \neg p'$: similar.
- $p \equiv p_1 \wedge p_2$: similar.
- $p \equiv \forall x.\, p'$: For all $n \in \mathbb{Z}$,

$$\sigma_1 := [\sigma \mid x : n]\,(w) \qquad \sigma_1' := \left[\sigma' \;\middle|\; x : n\right](w)$$

Then, $\forall w \in \mathrm{FV}\left(p'\right)$, $\sigma_1\,(w) = \sigma_1'\,(w)$. Therefore, by induction hypothesis,

$$\left[\!\left[p'\right]\!\right]\,\sigma_1 = \left[\!\left[p'\right]\!\right]\,\sigma_1'$$

$$\begin{aligned}\therefore [\![p]\!]\,\sigma &= \forall n \in \mathbb{Z}.\ \left[\!\left[p'\right]\!\right]\,\sigma_1 \\ &= \forall n \in \mathbb{Z}.\ \left[\!\left[p'\right]\!\right]\,\sigma_1' \\ &= [\![p]\!]\,\sigma'\end{aligned}$$

□

Property 1.2 (Substitution)**.**

$$\sigma\,(w) = \left[\![\delta\,(w)]\!\right]\,\sigma' \quad \Longrightarrow \quad [\![p/\delta]\!]\,\sigma' = [\![p]\!]\,\sigma$$

Proof. By structural induction. □

Property 1.3 (Finite substitution theorem♣)**.**

$$[\![p/v_0 \to e_0, \cdots, v_n \to e_n]\!]\,\sigma = [\![p]\!]\left[\sigma \;\middle|\; v_0 : [\![e_0]\!]\,\sigma \middle| \cdots \middle| v_n : [\![e_n]\!]\,\sigma\right]$$

Proof. By structural induction. □

Property 1.4 (Renaming♦)**.**

$$w \notin \mathrm{FV}_{\mathrm{assert}}\,(q) \setminus \{v\} \quad \Longrightarrow \quad [\![\forall w.\ q/v \to w]\!]\,\sigma = [\![\forall v.\ q]\!]\,\sigma$$

♣correspondence between syntactic and semantic substitutions.
♦renaming doesn't change the meaning of an assertion

Chapter 2

The Simple Imperative Language

2.1 Motivation or goal

(1) Most real-worlds PLs support computation by state update and that by function application. The former is often referred to as imperative computation, and the latter as functional or applicative computation Our goal is to study core PL concepts and ideas for imperative computation.

(2) Actually, it is more appropriate to say that our aim is a formal and mathematical analysis of the core PL concepts for imperative computation. We will study or learn mathematical tools for this. Also, we will show how to express and analyze big design decisions of such an imperative PLs.

(3) We will look at (i) some basic concepts and results of domain theory, (ii) variable declaration and binding, (iii) syntactic sugar error handling and (iv) the notions of soundness and full abstraction.(Well, I just listed all the key items in the chapter 2 of the book.)

(4) A good way to learn the material of this chapter is to ask yourself: What should you do in order to design an imperative programming language and build a foundation of the designed language? Think about this a little, and compare your answer with what I'll explain.

2.2 Syntax

(1) Variables, and read and update of them . . . key concepts or operations for imperative computation.

(2) Syntax that supports these concepts and operations:

$\langle intexp \rangle ::= 0 \mid 1 \mid 2 \mid \ldots \mid \langle var \rangle \mid \ldots$ [♣]

$\langle boolexp \rangle ::= \text{true} \mid \text{false} \mid \ldots$ [♦]

$\langle comm \rangle ::= \langle var \rangle := \langle intexp \rangle$ [♥] $\mid \text{skip} \mid \langle comm \rangle \,;\, \langle comm \rangle$ [♠]
$\mid$ if $\langle boolexp \rangle$ then $\langle comm \rangle$ else $\langle comm \rangle$
$\mid$ while $\langle boolexp \rangle$ do $\langle comm \rangle$

As in the case of predicate logic, you can understand $\langle comm \rangle$ as the set of all finite derivation trees, or as a multi-sorted initial algebra for the signature determined by the grammar.

(3) It is a language for expressing a sequence of variable reads and variable updates.

2.3 Baby domain theory

(1) Giving a denotational semantics to our simple imperative language is not as straightforward as doing so with predicate logic, because of loop.

(2) $[\![-]\!] : \langle comm \rangle \to \ldots$ [★]

We want the following equation for loop unrolling to hold:

$$\begin{aligned} [\![\text{while } b \text{ do } c]\!] &= [\![\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}]\!] \\ &= \ldots [\![\text{while } b \text{ do } c]\!] \ldots \\ &= F\left([\![\text{while } b \text{ do } c]\!]\right) \text{ for some } F. \end{aligned}$$

But a function F on a set may or may not have such a fixed point.

(3) Then, why should F in the above have a fixed point? Because it is something that can be implemented by a program.

$$F \text{ “} = \text{”} \text{ [❀] } [\![\text{if } b \text{ then } c; \square \text{ else skip}]\!] .$$

One objective of domain theory is to formalize good properties enjoyed by such program implementable functions without going into the low-level details of computability theory.

(4) High-level meta heuristic behind domain theory:

(i) Consider a set together with some structure.

(ii) Use functions between such sets that respect the structures.

[♣] same as before
[♦] almost the same as that of $\langle assert \rangle$. The exception is that $\langle boolexp \rangle$ doesn't include quantifiers. (Why?)
[♥] update of a variable
[♠] order matters
[★] don't worry about this target set now.
[❀] informally

(iii) Why on (i) and (ii)? Because if done well, functions in (ii) will always have fixed points.

(5) Key definitions:

Definition (partial order)**.** A binary relation $\sqsubseteq$ on a set S is a partial order if

(i) $x \sqsubseteq x$ for all $x \in S$ (reflexivity);

(ii) for all $x, y, z \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (transitivity); and

(iii) for all $x, y \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$ (anti-symmetry).

A set S with a partial order $\sqsubseteq$ is called a partially ordered set or poset.

Definition. A chain in a poset $(S, \sqsubseteq)$ is a (countably) infinite sequence

$$x_0, x_1, x_2, \ldots, x_n, \ldots$$

of elements in S s.t. $x_n \sqsubseteq x_{n+1}$ for all $n \geq 0$.

Definition. A pre-domain is a poset $(S, \sqsubseteq)$ such that all chains have least upper bounds. That is, for every chain $\{x_n\}_{n\geq 0}$ in S, there exists y ♣ in S s.t.

(i) ♦ $x_n \sqsubseteq y$ for all $n \geq 0$

(ii) ♥ for any z in S, if $x_n \sqsubseteq z$ for every $n \geq 0$, then $y \sqsubseteq z$.

Definition. A domain is a pre-domain $(S, \sqsubseteq)$ that has the least element, often denoted $\bot$. (meaning: for all $x \in S$, $\bot \sqsubseteq x$)

Definition. Let $(S_1, \sqsubseteq_1)$ and $(S_2, \sqsubseteq_2)$ be pre-domains. A function $f : S_1 \to S_2$ is continuous if for every chain $\{x_n\}_{n\geq 0}$ in S_1, $f\left(\bigsqcup_{n\geq 0} x_n\right)$ is the least upper bound of $\left\{f(x_n)\right\}_{n\geq 0}$ in S_2.

Definition. A function $f : S_1 \to S_2$ is monotone if for all $x, y \in S_1$,

$$x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y).$$

When S_1 and S_2 are domains with least elements $\bot_1$ and $\bot_2$, we say that a function $f : S_1 \to S_2$ is strict if $f(\bot_1) = \bot_2$.

Exercise 2.1. Show that if f is continuous, it is monotone.

(6) What's going on here? What are the intuitions behind these definitions?

(i) $x \sqsubseteq y$ ... intuitively means that y has more information than x or x and y have the same amount of information.

♣ notation for y: $\bigsqcup_{n\geq 0} x_n$

♦ y is an upper bound

♥ y is the least such

Example.

(a) $\mathbb{Z}^* \cup \mathbb{Z}^\omega$ ... finite or infinite sequence [♣] of integers. (without the infinite part, not a pre-domain) $x \sqsubseteq y$ iff x is an initial subsequence (or prefix) of y.

$$\langle 3,1,4\rangle \sqsubseteq \langle 3,1,4,1,5,9\rangle$$
$$\langle 3,1,4\rangle \not\sqsubseteq \langle 3,1,5,9\rangle$$

(b) $\mathbb{Z}_\perp \stackrel{\text{def}}{=} \mathbb{Z} \cup \{\perp\}$ ... lifted $\mathbb{Z}$. [♦]

$$\forall x, y \in \mathbb{Z}_\perp, \quad x \sqsubseteq y \iff x = \perp \text{ or } x = y.$$

(c) $\left(2^{\mathbb{Z}}, \subseteq\right)$ ... power set of $\mathbb{Z}$.

(d) vertical domain of natural numbers ...

$$\left.\begin{array}{r} \infty =: \top \\ \vdots \\ 2 \\ 1 \\ 0 =: \perp \end{array}\right\} =: \mathbb{N}^\top$$

(ii) The monotonicity means the preservation of the approximation relation.[♥] One intuition behind continuity of f is that in order to produce finite amount of information in its output, f consumes only finite amount information in its input.

Example.

(a)

$$\text{set} \in \left[Z^{*,\omega} \to 2^{\mathbb{Z}} \right]$$
$$\text{set}\left(\langle x_1, \ldots, x_n\rangle\right) = \{x_1, \ldots, x_n\}$$
$$\text{set}\left(\langle x_1, x_2, \ldots\rangle\right) = \{x_1, x_2, \ldots\}$$

If $A \subseteq \text{set}(s)$ and A is finite, then there is a finite prefix s_0 of s (i.e., $s_0 \sqsubseteq s$) s.t. $A \subseteq \text{set}(s_0)$.

Exercise 2.2.

Show that if $f \in \left[Z^{*,\omega} \to 2^{\mathbb{Z}} \right]$ is continuous, it satisfies the above property. Also, show that if $f \in \left[Z^{*,\omega} \to 2^{\mathbb{Z}} \right]$ is monotone and satisfies the property, then f is continuous.

[♣] results produced so far by sequence-producing computation.
[♦] integer output of an integer-returning computation
[♥] editor: if a is an approximation of b, then $f(a)$ is an approximation of $f(b)$

(b) $f \in \left[2^{\mathbb{Z}} \to \mathbb{N}^{\top}\right]$

$$f(A) = \begin{cases} |A| \text{ if A is finite} \\ \top \text{ if A is infinite} \end{cases}$$

Exercise 2.3.

A function from $2^{\mathbb{Z}}$ to a predomain P is finitely generated if for all $A \in 2^{\mathbb{Z}}$, $f(A)$ is the least upper bound of $\left\{f(A_0) \;\middle|\; A_0 \subseteq A \text{ and } A_0 \text{ is finite}\right\}$. Show that f is continuous iff it is finitely generated.

(iii) John Reynolds' phrase in page 108:

"... Instead, it is based on the physical ♣ *limitations of communication: one cannot predict the future of input, nor receive an infinite amount of information in a finite amount of time, nor produce output except at finite times ..."*

(7) One important reason of doing domain theory is to have the following "least fixed-point theorem":

Property 2.1 (Least Fixed-Point Theorem)**.** If D is a domain and f is a continuous function from D to D,

$$x = \bigsqcup_{n=0}^{\infty} f^n\left(\perp^{\blacklozenge}\right)$$

is the least fixed-point of f. (That is, $f(x) = x$ and for all $y \in D$ s.t. $f(y) = y$, $x \sqsubseteq y$.)

Proof. By Exercise 2.1, f is monotone. Using induction and $\perp$'s being the least element, we can show that $\left\{f^n(\perp)\right\}_{n\geq 0}$ is a chain in D. Since D is a domain, the least upper bound $\bigsqcup_{n=0}^{\infty} f^n(\perp)$ exists. Furthermore, by the continuity of f,

$$f(x) = f\left(\bigsqcup_{n=0}^{\infty} f^n(\perp)\right) = \bigsqcup_{n=0}^{\infty} f^{n+1}(\perp) = \bigsqcup_{n=0}^{\infty} f^n(\perp) = x.$$

$\therefore x$ is a fixed point of f.

To show that x is a least such, consider a fixed point y of f. Then, by induction, we can show that y is an upper bound of the chain $\left\{f^n(\perp)\right\}_{n\geq 0}$. $\therefore x \sqsubseteq y$. □

(8) When P, P' are predomains, we write $\left[P \underset{c}{\to} P'\right]$ for the set of continuous functions. When $\left[P \underset{c}{\to} P'\right]^{\blacktriangledown}$ is given pointwise order $\sqsubseteq$,

$$f \sqsubseteq g \iff f(x) \sqsubseteq_{P'} g(x) \text{ for all } x \in P, \quad f, g \in \left[P \underset{c}{\to} P'\right],$$

♣Domain theory attempts to capture this aspect of computation

♦least element of D

♥Although we rush here, this function space construction is very important. It lets domain theory be applicable to functional languages.

it becomes a predomain where the limit of a chain $\{f_n\}_{n\geq 0}$ is defined pointwise $x \mapsto \bigsqcup_{n=0}^{\infty} f_n(x)$. Furthermore, if P' is a domain with the least element $\bot'$, then $\left[P \underset{c}{\to} P'\right]$ is also a domain with $x \mapsto \bot'$ as its least element.

(9) D is a domain with the least element $\bot$. Define Y_D to be the following function from $\left[D \underset{c}{\to} D\right]$ to D:

$$Y_D(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot).$$

Lemma 2.1. Y_D is continuous♣.

Proof. Exercise. □

(10) There are a lot of interesting results in domain theory, some of which we will cover later in the course. Before finishing this mini review of domain theory, I want to explain the lifting construction.

(i)
- $P_\bot := P \cup \{\bot\}$ for a predomain P.
- $x \sqsubseteq_{P_\bot} y$ iff $x = \bot$ or $x, y \in P$ and $x \sqsubseteq_P y$ for $x, y \in P_\bot$.
- Intuitively, we are adding the least element to P and converting P to a domain.

(ii) $i_\uparrow \in \left[P \underset{c}{\to} P_\bot\right]$, sometimes called unit.

$$i_\uparrow(x) = x \text{ for all } x \in P.$$

(iii) For each $f \in \left[P \underset{c}{\to} P'_\bot\right]$,

$$f_{\shortparallel} \in \left[P_{\shortparallel} \underset{c}{\to} P'_{\shortparallel}\right]$$
$$f_{\shortparallel}(\bot) = \bot$$
$$f_{\shortparallel}(x) = f(x) \text{ for all } x \in P.$$

sometimes called Kleisli extension.

(iv) Why should we care about (ii) and (iii)? Because they allow us to compose continuous functions from P to $P'_\bot$

$$f \in \left[P \underset{c}{\to} P'_\bot\right] \text{ and } g \in \left[P' \underset{c}{\to} P''_\bot\right] \implies g_{\shortparallel} \circ f \in \left[P \underset{c}{\to} P''_\bot\right].$$

We can view $(-)_{\shortparallel} \circ (-)$ as a new composition operator $\circ'$. Then, $\circ'$ is associative and $i_\uparrow \circ' f = f = f \circ' i_\uparrow$. This means that $(-_\bot, i_\uparrow, -_{\shortparallel})$ gives rise to a monad on predomains.

♣This means that the very act of computing a fixed point of a given function is continuous.

2.4 Denotational semantics of the simple imperative language

(1) Recall that $\Sigma = \langle var \rangle \to \mathbb{Z}$. Σ is a predomain when given the discrete order $\sqsubseteq$. (That is, $x \sqsubseteq y$ iff $x = y$ for all $x, y \in \Sigma$.)

$$[\![-]\!]_{\text{intexp}} \in \langle intexp \rangle \to [\,\Sigma \to \mathbb{Z}\,]\,(sameasbefore)$$
$$[\![-]\!]_{\text{boolexp}} \in \langle boolexp \rangle \to [\,\Sigma \to \mathbb{B}\,]\,(sameasbefore)$$
$$[\![-]\!]_{\text{comm}} \in \langle comm \rangle \to \left[\, \Sigma \underset{c}{\to} \Sigma_\perp \,\right]$$

$$[\![x := e]\!]\,\sigma = \left[\sigma \;\middle|\; x : [\![e]\!]\,\sigma\right]$$
$$[\![\text{skip}]\!]\,\sigma = \sigma$$
$$[\![c_1;\ c_2]\!]\,\sigma = [\![c_2]\!]_{\perp\!\!\perp}\left([\![c_1]\!]\,\sigma\right)$$
$$[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!]\,\sigma = \text{if } [\![b]\!]\,\sigma \text{ then } [\![c_1]\!]\,\sigma \text{ else } [\![c_2]\!]\,\sigma$$
$$[\![\text{while } b \text{ do } c]\!]\,\sigma = Y_{\Sigma_\perp}(F)$$
$$\text{where } F \in \left[\, \Sigma \underset{c}{\to} \Sigma_\perp \,\right] \underset{c}{\to} \left[\, \Sigma \underset{c}{\to} \Sigma_\perp \,\right]$$
$$\text{and } F(f)(\sigma) := \text{if } [\![b]\!]\,\sigma \text{ then } \left(f_{\perp\!\!\perp} \circ [\![c]\!]\right)(\sigma) \text{ else } \sigma$$

Note: $Y_{\Sigma_\perp}$ and $\underset{c}{\to}$ are where we get something by using domain theory.

(2) Why <u>least</u> fixed point? Because the least fixed point maps an input state to $\perp$ (denoting non-termination, hence, the absence of any information) whenever the corresponding output state is not uniquely determined by the equation $F(f) = f$. For example,

(i) least fixed point ... $[\![\text{while true do skip}]\!]\,\sigma = \perp$.

(ii) non-least fixed point ... $[\![\text{while true do skip}]\!]\,\sigma = \sigma$.

$$\begin{aligned} F(f)(\sigma) &= \text{if } [\![\text{true}]\!]\,\sigma \text{ then } \left(f_{\perp\!\!\perp} \circ [\![\text{skip}]\!]\right)(\sigma) \text{ else } \sigma \\ &= \left(f_{\perp\!\!\perp} \circ [\![\text{skip}]\!]\right)(\sigma) \\ &= f_{\perp\!\!\perp}\left([\![\text{skip}]\!]\,(\sigma)\right) \\ &= f_{\perp\!\!\perp}\,(\sigma) \\ &= f\,(\sigma) \end{aligned}$$

That is, $F(f) = f$.

Later when we consider the correspondence between denotational semantics and operational semantics, we will answer this question more rigorously.

(3) Design decision of our language seen in the semantics:

$$[\![-]\!]_{\text{intexp}} : \langle intexp \rangle \to \Sigma \to \mathbb{Z}$$

all integer expressions terminate and do not raise exceptions. A similar remark applies to boolean expressions as well.

(4) Choosing the type of the semantics function such as $\langle intexp \rangle \to \Sigma \to \mathbb{Z}$ is the most important step in defining the semantics. It also clarifies certain major design decisions of the target programming language.

2.5 Variable declaration and substitution

(1) $\langle comm \rangle ::= \ldots \mid \text{newvar } \langle var \rangle := \langle intexp \rangle \text{ in } \langle comm \rangle$

This is a construct that doesn't increase the expressivity of the language but enables the programmers to combat the complexity of software by introducing the ideas of scope and local variables.

(2)

$$[\![\text{newvar } v := e \text{ in } c]\!]\,\sigma = \Big(\big(\lambda\sigma' \in \Sigma\,.\ \big[\sigma' \;\big|\; v : \sigma v\big]\big)_{\perp\!\!\perp} \circ [\![c]\!]\Big)\Big(\big[\sigma \;\big|\; v : [\![e]\!]\,\sigma\big]\Big)$$

$\ldots\, \lambda\sigma' \in \Sigma\,.\ \big[\sigma' \;\big|\; v : \sigma v\big]$ means the function $\sigma' \mapsto \big[\sigma' \;\big|\; v : \sigma v\big]$, restoring the old value. We didn't have to do something like this when we interpreted quantifications in predicate logic. This is because there we didn't return a state, but a boolean value.

(3) How do we know that this is a sensible definition? By checking expected properties like Property 2.2 and Property 2.3.

$\mathrm{FV}\,(c)$ ... free variables appearing in c. (textbook page 40)

$\mathrm{FA}\,(c)$ ... free assigned variables appearing in c. (textbook page 41)

Property 2.2 (Coincidence)**.**

(a) $\sigma w = \sigma' w$ for all $w \in \mathrm{FV}\,(c)$

$$\implies \big([\![c]\!]\,\sigma = [\![c]\!]\,\sigma' = \perp\big) \text{ or } \Big([\![c]\!]\,\sigma, [\![c]\!]\,\sigma' \in \Sigma \text{ and } \big([\![c]\!]\,\sigma\big)\,w = \big([\![c]\!]\,\sigma'\big)\,w \text{ for all } w \in \mathrm{FV}\,(c)\Big)$$

(b) $[\![c]\!]\,\sigma \neq \perp \implies \big([\![c]\!]\,\sigma\big)\,w = \sigma w$ for all $w \notin \mathrm{FA}\,(c)$.

Property 2.3 (Renaming)**.**

$$v_{\text{new}} \notin \mathrm{FV}\,\big(c'\big) - \{v\} \implies \Big[\!\Big[\text{newvar } v := e \text{ in } c'\Big]\!\Big]\,\sigma = \Big[\!\Big[\text{newvar } v_{\text{new}} := e \text{ in } c'/_{v \to v_{\text{new}}}\Big]\!\Big]\,\sigma$$

2.6 Syntactic Sugar

(1) Introduction of a construct by defining its meaning in terms of existing constructs in the language.

(2) Three definitions of for loop:

(i) $(\text{for } v := e_0 \text{ to } e_1 \text{ do } c) := \big(v := e_0;\ \text{while } v \leq e_1 \text{ do } (c;\ v := v + 1)\big)$

(ii) $(\text{for } v := e_0 \text{ to } e_1 \text{ do } c) := \big(\text{newvar } v := e_0 \text{ in while } v \leq e_1 \text{ do } (c;\ v := v + 1)\big)$

(iii) $(\text{for } v := e_0 \text{ to } e_1 \text{ do } c) :=$
$\big(\text{newvar } w := e_1 \text{ in newvar } v := e_0 \text{ in while } v \leq w \text{ do } (c;\ v := v + 1)\big)$,
where $w \neq v$ and $w \notin \mathrm{FV}(e_0) \cup \mathrm{FV}(c)$.

(iv) (iii) with the condition $v \notin \mathrm{FV}(c)$.

(3) The for loop should be something easier to understand than while. In this regard, (i) < (ii) < (iii) < (iv).

2.7 Arithmetic errors

(1) How should we deal with $x \div 0$, underflow and overflow?

(2) Two approaches:

(i) early stop with error

(ii) some default choice and computation continued: ad hoc but it can become less ad hoc if we ensure that the default choices satisfy certain properties such as

$$[\![(x + y) \times 0]\!]\,\sigma = 0$$
$$[\![x \div 0 = x \div 0]\!]\,\sigma = \mathbf{tt}$$
$$[\![y := x \div 0;\ y := e]\!]\,\sigma = [\![y := e]\!]\,\sigma \text{ when } y \notin \mathrm{FV}(e)$$
$$[\![\text{if } x + y = z \text{ then } c \text{ else } c]\!]\,\sigma = [\![c]\!]\,\sigma.$$

2.8 Soundness and full abstraction

(1) The semantics defined so far looks ok, but is there any formal way to confirm this?

(2) One approach is to show that the semantics assigns the same meaning to two commands c_1 and c_2 only when c_1 and c_2 should be equal intuitively. That is,

$$[\![c_1]\!] = [\![c_2]\!] \implies c_1 \text{“} = \text{”}^{\clubsuit} c_2.$$

This property is called soundness. Its converse

$$[\![c_1]\!] = [\![c_2]\!] \impliedby c_1 \text{“} = \text{”} c_2$$

is called full abstraction.

$^{\clubsuit}$our intuitive notion of equality defined separately

(3) Now how to define " = "? We use a set of observable phrases with a hole or contexts, $\mathcal{C}$, and a set of observations, $\mathcal{O}$, which are functions from observable phrases to outcomes.

$$c_1 \text{“} = \text{”} c_2 \iff \forall c \in \mathcal{C}^{\clubsuit},\ \forall o \in \mathcal{O}^{\blacklozenge},\ o\left(c\left[\,c_1\,\right]^{\blacktriangledown}\right) = o\left(c\left[\,c_2\,\right]\right)$$

(i) Intuitively, this condition says that under all use cases, the user cannot observe the difference between c_1 and c_2.

(ii) This is sometimes called observataional equivalence.

(iii) Note that this is not a syntax-directed (or compositional) definition.

Example. Assume that $v_0, \ldots, v_{n-1}$ are all the free variables in c_1 and c_2.

$$\mathcal{C} = \left\{ \begin{array}{c} \text{newvar } v_0 := k_0 \text{ in} \\ \text{newvar } v_1 := k_1 \text{ in} \\ \vdots \\ \text{newvar } v_{n-1} := k_{n-1} \text{ in} \\ \left([-];\ \text{if } v_i = k \text{ then skip} \right. \\ \left. \qquad\qquad \text{else while true do skip} \right) \end{array} \middle| \begin{array}{c} \\ \\ \\ k, k_0, \ldots, k_{n-1} \in \mathbb{Z} \\ i \in \{0, \ldots, n-1\} \\ \\ \end{array} \right\}$$

$$\mathcal{O} = \left\{ \lambda c \,.\ \text{if } [\![c]\!]\,\sigma_0 = \perp \text{ then } 0 \text{ else } 1 \right\}$$

where $\sigma_0\,(x) = 0$ for all x.

$^{\clubsuit}$intuitively means all use cases
$^{\blacklozenge}$intuitively means user's observations
$^{\blacktriangledown}$filling the hole of c with c_1

Chapter 3

Program Specifications and Their Proofs

3.1 Motivation

(1) Are there any methods that let us specify desired properties or intended behaviors of a program and prove that the specified properties indeed hold? For instance, consider the program:

$$c_{\text{div3}} = \left(\begin{array}{l} a := 0; \\ b := x; \\ \text{while } (b \geq 3) \text{ do} \\ \quad\quad b := b - 3; \\ \quad\quad a := a + 1 \end{array} \right)$$

We want to express formally that the program divides x by 3 and stores the quotient in a and the remainder in b. We also want to prove this formal specification.

(2) We will study Hoare logic and its variant for total correctness. They provide the kind of methods that we are looking for.

(3) Hoare logic and its total-correctness variant are the basis of modern automatic software verifiers, such as Facebook's infer.

(4) Another reason for studying Hoare logic is that it shows why we need or where we use denotational semantics that we studied.

3.2 Syntax and semantics of specifications

(1) Specifications are a new type of phrases that formally express properties of programs.

(2) Syntax in terms of abstract grammar:

$$\begin{array}{rcl} \langle \mathit{spec} \rangle & ::= & \{\langle \mathit{assert} \rangle\}\ \langle \mathit{comm} \rangle\ \{\langle \mathit{assert} \rangle\} \\ & | & [\langle \mathit{assert} \rangle]\ \langle \mathit{comm} \rangle\ [\langle \mathit{assert} \rangle] \end{array}$$

Example.

$$c_{\text{fib}} = \left(\begin{array}{c} k := 1;\ y := 0;\ x := 1; \\ \text{while } k \neq n \text{ do} \\ (t := y;\ y := x;\ x := x + t;\ k := k + 1) \end{array} \right)$$

$$\{n \geq 0\}\, c_{\text{fib}}\, \big\{x = \text{fib}\,(n)\big\} \qquad \{\text{true}\}\, c_{\text{fib}}\, \big\{x = \text{fib}\,(n)\big\}$$

$$[\, n \geq 0\,]\, c_{\text{fib}}\, \big[\, x = \text{fib}\,(n)\,\big] \qquad [\,\text{true}\,]\, c_{\text{fib}}\, \big[\, x = \text{fib}\,(n)\,\big]$$

$$\{x \geq 0\}\, c_{\text{div3}}\, \{x = 3a + b \wedge 0 \leq b < 3 \wedge a \geq 0\}$$

$$[\, x \geq 0\,]\, c_{\text{div3}}\, [\, x = 3a + b \wedge 0 \leq b < 3 \wedge a \geq 0\,]$$

$$[\,\text{true}\,]\, c_{\text{div3}}\, [\, x = 3a + b \wedge 0 \leq b < 3 \wedge a \geq 0\,]$$

$$[\,\text{true}\,]\, c_{\text{div3}}\, [\, x = 3a + b \wedge 0 \leq b < 3 \wedge a \geq 0\,]$$

(3) Intuitive reading:

(i) $\{p\}\, c\, \{q\}$ holds iff when c is run in a state satisfying p, and it terminates normally, ♣ then the final state satisfies q.

(ii) $[\, p\,]\, c\, [\, q\,]$ holds iff when c is run in a state satisfying p, then it terminates normally, ♦ and the final state satisfies q.

Note that $[\, p\,]\, c\, [\, q\,]$ expresses a stronger property than $\{p\}\, c\, \{q\}$. The former is called total correctness specification, ♥ and the latter partial correctness specification. ♠

p ... precondition or precedent.

q ... postcondition or consequent.

Exercise 3.1. Among all the partial correctness and total correctness specification from above, pick those that hold.

(4) Formal semantics:

$$[\![-]\!] \in \big[\, \langle \mathit{spec} \rangle \to \mathbb{B}\,\big]$$

$$[\![\{p\}\, c\, \{q\}]\!] = \mathbf{tt} \quad \text{iff} \quad [\![p]\!]\, \sigma = \mathbf{tt} \wedge [\![c]\!]\, \sigma \neq \bot \implies [\![q]\!]\, \big([\![c]\!]\, \sigma\big) = \mathbf{tt}$$

$$[\![[\, p\,]\, c\, [\, q\,]]\!] = \mathbf{tt} \quad \text{iff} \quad [\![p]\!]\, \sigma = \mathbf{tt} \implies [\![c]\!]\, \sigma \neq \bot \wedge [\![q]\!]\, \big([\![c]\!]\, \sigma\big) = \mathbf{tt}$$

3.3 Inference rules

(1) Methods or rules for proving or deriving partial or total correctness triples.

(2)

$$\frac{\text{premises}}{\text{conclusion}} \qquad \frac{\varphi_1 \quad \varphi_2 \quad \ldots \quad \varphi_n}{\psi}$$

(if $\varphi_1, \varphi_2, \ldots, \varphi_n$ are true, then ψ is true)

♣ condition
♦ conclusion
♥ also total correctness triple
♠ also partial correctness triple, Hoare triple or triple

(3) Rules associated with program constructs:

$$\frac{}{\{p\}\,\text{skip}\,\{p\}} \qquad \frac{}{[\,p\,]\,\text{skip}\,[\,p\,]}$$

$$\frac{\{p\}\,c_1\,\{r\} \qquad \{r\}\,c_2\,\{q\}}{\{p\}\,c_1; c_2\,\{q\}} \qquad \frac{[\,p\,]\,c_1\,[\,r\,] \qquad [\,r\,]\,c_2\,[\,q\,]}{[\,p\,]\,c_1; c_2\,[\,q\,]}$$

$$\frac{\{p \wedge b\}\,c_1\,\{q\} \qquad \{p \wedge \neg b\}\,c_2\,\{q\}}{\{p\}\,\text{if } b \text{ then } c_1 \text{ else } c_2\,\{q\}} \qquad \frac{[\,p \wedge b\,]\,c_1\,[\,q\,] \qquad [\,p \wedge \neg b\,]\,c_2\,[\,q\,]}{[\,p\,]\,\text{if } b \text{ then } c_1 \text{ else } c_2\,[\,q\,]}$$

$$\frac{}{\left\{q/_{x \to e}\right\}\,x := e\,\{q\}} \qquad \frac{}{\left[\,q/_{x \to e}\,\right]\,x := e\,[\,q\,]}$$

$$\frac{\{i \wedge b\}\,c\,\{i\}}{\{i\}\,\text{while } b \text{ do } c\,\{i \wedge \neg b\}} \qquad \frac{i \wedge b \Rightarrow e \geq 0^{\blacklozenge} \qquad \left[\,i \wedge b \wedge e = v_0{}^{\clubsuit}\,\right]\,c\,[\,i \wedge e < v_0\,]}{[\,i\,]\,\text{while } b \text{ do } c\,[\,i \wedge \neg b\,]}$$

(i) Note that rules for total correctness and the corresponding ones for partial correctness are identical except for the case of loop. This is expected because these two notions differ only in their treatment of non-termination.

(ii) The rules for while require that i should be preserved by the body of the loop. The one for total correctness additionally requires that the value of e should decrease whenever we run the loop body c once, but it cannot be negative. All these requirements together give the conclusions of the rules.

(iii) The rules for assignment also deserve some thoughts. They in a sense say that running an assignment backward symbolically is the same as doing substitution. It holds because of the substitution theorem (Prop 1.3 and Prop 1.4 in the textbook).

Reminder of the theorem specialized to our case here:

$$\begin{array}{ccc} \Sigma & \xrightarrow{\lambda\sigma.\left[\sigma \mid x : [\![e]\!]\sigma\right]} & \Sigma \\ {\scriptstyle [\![q/_{x \to e}]\!]}\downarrow & & \downarrow{\scriptstyle [\![q]\!]} \\ \mathbb{B} & = & \mathbb{B} \end{array}$$

Rules not associated with any specific program constructs (sometimes called structural rules or adaptation rules):

$$\frac{p \Rightarrow p' \qquad \left\{p'\right\}\,c\,\left\{q'\right\} \qquad q' \Rightarrow q}{\{p\}\,c\,\{q\}} \qquad \frac{p \Rightarrow p' \qquad \left[\,p'\,\right]\,c\,\left[\,q'\,\right] \qquad q' \Rightarrow q}{[\,p\,]\,c\,[\,q\,]}$$

$^{\clubsuit}i \wedge b \Rightarrow e \geq 0$ should be valid. That is, $[\![i \wedge b \Rightarrow e \geq 0]\!]\,\sigma = \mathbf{tt}$ for all σ.

$^{\blacklozenge}$when v_0 does not occur free in i, b, c or e

They are called the rule of consequence. They often enable us to use the other rules, in particular, those for loop and if.

(4) I omit many structural rules and the rule for newvar. Look at the textbook if you are interested.

3.4 Example proof

$$c_{\text{div3}} = \left(\begin{array}{l} a := 0; \\ b := x; \\ \text{while } (b \geq 3) \text{ do} \\ \quad b := b - 3; \\ \quad a := a + 1 \end{array} \right)$$

Goal: prove that

$$\{x \geq 0\}\, c_{\text{div3}}\, \{x = 3a + b \wedge 0 \leq b < 3\}$$

Proof:

$$\dfrac{x \geq 0 \Rightarrow x = 3 \times 0 + x \wedge x \geq 0 \qquad \dfrac{\overline{\{x = 3 \times 0 + x \wedge x \geq 0\}\ a := 0\ \{x = 3a + x \wedge x \geq 0\}}}{\{x \geq 0\}\ a := 0\ \{x = 3a + x \wedge x \geq 0\}} \qquad \dfrac{}{\{x = 3a + x \wedge x \geq 0\}\ b := x\ \{x = 3a + b \wedge b \geq 0\}}}{\{x \geq 0\}\ a := 0;\ b := x\ \{x = 3a + b \wedge b \geq 0\}}$$

$$\dfrac{\dfrac{\dfrac{x = 3a + b \wedge b \geq 0 \wedge b \geq 3}{\Downarrow} \atop x = 3(a+1) + (b-3) \wedge b - 3 \geq 0 \qquad \overline{\left\{ x = 3(a+1) + (b-3) \wedge b - 3 \geq 0 \right\}\ b := b - 3\ \left\{ x = 3(a+1) + b \wedge b \geq 0 \right\}}}{\{x = 3a + b \wedge b \geq 0 \wedge b \geq 3\}\ b := b - 3\ \left\{ x = 3(a+1) + b \wedge b \geq 0 \right\}} \qquad \dfrac{}{\left\{ x = 3(a+1) + b \wedge b \geq 0 \right\}\ a := a + 1\ \{x = 3a + b \wedge b \geq 0\}}}{\{x = 3a + b \wedge b \geq 0 \wedge b \geq 3\}\ b := b - 3;\ a := a + 1\ \{x = 3a + b \wedge b \geq 0\}}$$

$$\dfrac{}{\{x = 3a + b \wedge b \geq 0\}\ \text{while } b \geq 3 \text{ do } b := b - 3;\ a := a + 1\ \{x = 3a + b \wedge 0 \leq b < 3\}}$$

$$\dfrac{\{x \geq 0\}\ a := 0;\ b := x\ \{x = 3a + b \wedge b \geq 0\} \qquad \{x = 3a + b \wedge b \geq 0\}\ \text{while } b \geq 3 \text{ do } b := b - 3;\ a := a + 1\ \{x = 3a + b \wedge 0 \leq b < 3\}}{\{x \geq 0\}\ c_{\text{div3}}\ \{x = 3a + b \wedge 0 \leq b < 3\}}$$

(1) In practice, people use the rule of consequence, without mentioning it explicitly. Also, they use many derived rules.

(2) This proof has the flavor of running a program backward symbolically because of its heavy use of the assignment rule and the fact that the rule of consequence is used only when it is necessary.

Exercise 3.2.

Prove:

(a)

$$\{n \geq 1\}\, c_{\text{fib}}\, \left\{ x = \text{fib}\,(n) \right\}$$

(b)

$$c_{\text{Euclid}} = \left(\begin{array}{l} \textit{while}\,(a \neq b) \text{ do} \\ \quad \text{if } a > b \text{ then } a := a - b \\ \quad \text{else } b := b - a \end{array} \right)$$

$$\{a \geq 1 \wedge b \geq 1 \wedge a = a_0 \wedge b = b_0\}\, c_{\text{Euclid}}\, \left\{ a = \gcd\,(a_0, b_0) \right\}$$

Exercise 3.3.

Find a forward rule for assignment. That is, for all p and x, e, find q s.t.

$$\frac{}{[\,p\,]\,x := e\,[\,q\,]}$$

3.5 Soundness

Theorem 3.1 (Soundness theorem)**.** If $[\,p\,]\,c\,[\,q\,]$ is derivable using the rules that we studied, [♣] then $[\![\{p\}\,c\,\{q\}]\!] = \mathbf{tt}$, i.e., the triple $\{p\}\,c\,\{q\}$ holds. If $[\,p\,]\,c\,[\,q\,]$ is derivable, then $[\![[\,p\,]\,c\,[\,q\,]]\!] = \mathbf{tt}$.

Proof. Intuitively, the theorem says that all rules are correct. In fact, typical proofs of the theorem show the correctness of the rules in the following sense:

If $\dfrac{\varphi_1 \quad \varphi_2 \quad \cdots \quad \varphi_n}{\psi}$ then

$$[\![\varphi_1]\!] = \mathbf{tt} \wedge [\![\varphi_2]\!] = \mathbf{tt} \wedge \cdots \wedge [\![\varphi_n]\!] = \mathbf{tt} \implies [\![\psi]\!] = \mathbf{tt}.$$

The rules for loop (or while) are the most important cases. We will consider only the one for partial correctness.

$$\frac{\{i \wedge b\}\,c\,\{i\}}{\{i\}\,\text{while } b \text{ do } c\,\{i \wedge \neg b\}}$$

We first do a bit of rewriting for the semantics of specifications.

$[\![[\,p\,]\,c\,[\,q\,]]\!]$ iff

$$\forall \sigma \in \Sigma,\ [\![p]\!]\,\sigma = \mathbf{tt} \implies [\![q]\!]_\perp \left([\![c]\!]\,\sigma\right) \sqsubseteq \mathbf{tt}$$

$$\text{where } [\![q]\!]_\perp \in [\,\Sigma_\perp \to \mathbb{B}_\perp\,]$$

$$\text{s.t. } [\![q]\!]_\perp\,\sigma = [\![q]\!]\,\sigma \text{ for all } \sigma \in \Sigma \text{ and } [\![q]\!]_\perp\,(\perp) = \perp$$

We need to prove that, if

$$\forall \sigma.\ [\![i \wedge b]\!]\,\sigma = \mathbf{tt} \implies [\![i]\!]_\perp \left([\![c]\!]\,\sigma\right) \sqsubseteq \mathbf{tt} \qquad (\star)$$

then

$$\forall \sigma.\ [\![i]\!]\,\sigma = \mathbf{tt} \implies [\![i \wedge \neg b]\!]_\perp \left([\![\text{while } b \text{ do } c]\!]\,\sigma\right) \sqsubseteq \mathbf{tt}$$

Assume that Eq. $(\star)$ holds. Let

$$F \in \left[(\Sigma \to \Sigma_\perp) \underset{c}{\to} (\Sigma \to \Sigma_\perp) \right]$$

$$F\,(f)\,(\sigma) = \text{if } [\![b]\!]\,\sigma = \mathbf{tt} \text{ then } \left(f_\perp \circ [\![c]\!]\right)(\sigma) \text{ else } \sigma.$$

Define $f_n := F^n\,(\perp)$ for all $n \geq 0$. Then, $[\![\text{while } b \text{ do } c]\!] = \bigcup_{n=0}^{\infty} f_n$. We will show that for all $n \geq 0$,

$$\forall \sigma.\ [\![i]\!]\,\sigma = \mathbf{tt} \implies [\![i \wedge \neg b]\!]_\perp \left(f_n\,(\sigma)\right) \sqsubseteq \mathbf{tt}. \qquad (\star\star)$$

[♣] called rules in Hoare logic

This is sufficient because for all $\sigma \in \Sigma$ s.t. $[\![i]\!]\,\sigma = \mathbf{tt}$,

$$\begin{aligned}
[\![i \wedge \neg b]\!]_\perp \left([\![\text{while } b \text{ do } c]\!]\,\sigma\right) &= [\![i \wedge \neg b]\!]_\perp \left(\left(\bigcup_{n=0}^{\infty} f_n\right)(\sigma)\right) \\
&= [\![i \wedge \neg b]\!]_\perp \left(\bigcup_{n=0}^{\infty} f_n\,(\sigma)\right) \\
&=^{\clubsuit} \bigcup_{n=0}^{\infty} [\![i \wedge \neg b]\!]_\perp \left(f_n\,(\sigma)\right) \\
&\sqsubseteq^{\blacklozenge} \bigcup_{n=0}^{\infty} \mathbf{tt} = \mathbf{tt}.
\end{aligned}$$

Our proof of Eq. $(\star\star)$ uses induction on n.

- Base case $n = 0$: $f_0 = \perp$.

$$\therefore [\![i \wedge \neg b]\!]_\perp \left(f_0\,(\sigma)\right) = [\![i \wedge \neg b]\!]_\perp (\perp) = \perp \sqsubseteq \mathbf{tt}.$$

- Inductive case $n = m + 1$: Pick σ s.t. $[\![i]\!]\,\sigma = \mathbf{tt}$.

$$\begin{aligned}
&[\![i \wedge \neg b]\!]_\perp \left(f_{m+1}\,(\sigma)\right) \\
&= [\![i \wedge \neg b]\!]_\perp \left(F\,(f_m)\,(\sigma)\right) \\
&= [\![i \wedge \neg b]\!]_\perp \left(\text{if } [\![b]\!]\,\sigma = \mathbf{tt} \text{ then } \left(f_{m\,\perp\!\!\!\perp} \circ [\![c]\!]\right)(\sigma) \text{ else } \sigma\right) \\
&= \text{if } [\![b]\!]\,\sigma = \mathbf{tt} \text{ then } \left([\![i \wedge \neg b]\!]_\perp \circ f_{m\,\perp\!\!\!\perp}\right)\left([\![c]\!]\,(\sigma)\right) \text{ else } [\![i \wedge \neg b]\!]_\perp (\sigma)
\end{aligned}$$

  Since $[\![i]\!]\,\sigma = \mathbf{tt}$, if $[\![b]\!]\,\sigma \neq \mathbf{tt}$, then

$$[\![i \wedge \neg b]\!]_\perp (\sigma) = [\![i]\!]_\perp (\sigma) = \mathbf{tt} \sqsubseteq \mathbf{tt}.$$

  If $[\![b]\!]\,\sigma = \mathbf{tt}$ and $[\![c]\!]\,\sigma = \perp$, then

$$\left([\![i \wedge \neg b]\!]_\perp \circ f_{m\,\perp\!\!\!\perp}\right)\left([\![c]\!]\,(\sigma)\right) = \perp \sqsubseteq \mathbf{tt}.$$

  If $[\![b]\!]\,\sigma = \mathbf{tt}$ and $[\![c]\!]\,\sigma \neq \perp$, then $[\![i]\!]_\perp \left([\![c]\!]\,\sigma\right) = \mathbf{tt}$ by Eq. $(\star)$. Thus,

$$\left([\![i \wedge \neg b]\!]_\perp \circ f_{m\,\perp\!\!\!\perp}\right)\left([\![c]\!]\,(\sigma)\right) = [\![i \wedge \neg b]\!]_\perp \left(f_m\left([\![c]\!]\,(\sigma)\right)\right) \sqsubseteq \mathbf{tt}$$

  by induction hypothesis.

□

$^{\clubsuit}$because $[\![i \wedge \neg b]\!]_\perp$ is continuous
$^{\blacklozenge}$because of Eq. $(\star\star)$

Chapter 4

Failure, Input-Output, and Continuations

4.1 Motivation

(1) Realistic programming languages have a wide range of language constructs and features. In particular, they have constructs for input and output and those for altering the flow of exectuion.

(2) We will study mathematical tools that allow us to study or analyze constructs for input-output and the fail operation, the simplest operator for changing the flow of execution.

(3) Specifically, we will study recursively defined domains, the disjoint union of domains and continuations, and use them to define denotational semantics of an imperative language with input-output and failure.

(4) Here are a few transferable high-level messages that I want you to learn in the chapter.

(i) Recursively defined domains can be used to model computations with intermediate results[♣] and computations that may be stopped in the middle and be resumed later[♦].

(ii) Continuations may simplify semantic definitions by providing a canonical treatment of sequential composition operator.

4.2 Syntax of a programming language with failure and input-output

$\langle comm \rangle ::= \ldots \mid \text{newvar } \langle var \rangle := \langle intexp \rangle \text{ in } \langle comm \rangle \mid \text{fail}^{\blacktriangledown} \mid ?\langle var \rangle^{\spadesuit} \mid !\langle intexp \rangle^{\bigstar}$

[♣] such as computations with outputs
[♦] such as computations with inputs
[▼] failure
[♠] input
[★] output

Fail terminates the current execution and makes the current state the final state of execution modulo the restoration of values of global variables.

Exercise 4.1. Write two interesting programs in this language.

Exercise 4.2. What should be the final state of the following program?

```
x := 124; y := 0;
(newvar x := 3 in fail);
y := 2
```

4.3 Semantics

(1) The most important step in defining the semantics is to decide the form of the interpretation function for commands:

$$[\![-]\!] \in \left[\langle comm \rangle \to \left[\underline{A} \underset{c}{\overset{\clubsuit}{\to}} \underline{B} \right] \right]$$

What predomains or domains should we use for $\underline{A}$ and $\underline{B}$?

(2) $\underline{A}$ should be $\Sigma = [\, Var \to \mathbb{Z}\,]$, the set of (or predomain) of states.$^{\blacklozenge}$

(3) $\underline{B}$ is complex. Its element describe computations that can be stopped and resumed, and may output some integers in the middle. Formally,

$$\underline{B} = \Omega \simeq^{\blacktriangledown} \left(\hat{\Sigma} + (\mathbb{Z} \times \Omega) + \left(\mathbb{Z} \underset{c}{\to} \Omega \right) \right)_{\perp}^{\spadesuit\bigstar}$$

$$\hat{\Sigma} = \Sigma \cup \{\text{abort}\} \times \Sigma \simeq \Sigma + \Sigma$$

(4) Many things here are not defined nor explained. We will look at them one by one. But before doing so, let's try to understand intuitions behind this definition.

(i) non-termination $\ldots \perp$

(ii) normal termination with a state $\sigma \ldots \sigma \in \Sigma$

(iii) abnormal termination with a state $\sigma \ldots \langle \text{abort}, \sigma \rangle \in \{\text{abort}\} \times \Sigma$

(iv) output n and the rest of computation $\omega \ldots \langle n, \omega \rangle \in \mathbb{Z} \times \Omega$

(v) suspended computation g that waits for an input $\ldots g \in \left(\mathbb{Z} \underset{c}{\to} \Omega \right)$

$^{\clubsuit}$continuous function

$^{\blacklozenge}$Recall that a set can be viewed as a predomain when it is given $=$ as its order $\sqsubseteq$, which is called discrete order.

$^{\blacktriangledown}$is isomorphic to

$^{\spadesuit}\Omega$ satisfies a form of equation

$^{\bigstar}$The RHS of the isomorphism says that there are five kinds of outcomes of running a command at a given state. We list these cases below.

(5) The description of Ω uses two pre-domain constructors, sum $+$ and product $\times$. Let $P_0, \cdots, P_{n-1}$ be pre-domains and let $\sqsubseteq_0, \cdots, \sqsubseteq_{n-1}$ be the partial orders of these pre-domains. From these pre-domains, we can construct the following two pre-domains, their sum and product:

$$P_0 + \cdots + P_{n-1} = \left\{ \langle i, x \rangle \;\middle|\; i \in \{0, \cdots, n-1\}, x \in P_i \right\}$$
$$\langle i, x \rangle \sqsubseteq \langle j, y \rangle \text{ iff } i = j \text{ and } x \sqsubseteq_i y$$

$$P_0 \times \cdots \times P_{n-1} = \left\{ \langle x_0, \cdots, x_{n-1} \rangle \;\middle|\; x_i \in P_i \right\}$$
$$\langle x_0, \cdots, x_{n-1} \rangle \sqsubseteq \langle y_0, \cdots, y_{n-1} \rangle \text{ iff } x_i \sqsubseteq_i y_i \text{ for all } i \in \{0, \cdots, n-1\}$$

(i)

Exercise 4.3.

Show that $P_0 + \cdots + P_{n-1}$ and $P_0 \times \cdots \times P_{n-1}$ are pre-domains. Also, prove that $P_0 \times \cdots \times P_{n-1}$ is a domain if all P_i's are domains.

Hint/Information:

(a) The least upper bound of a chain $\left\{ \left\langle x_0^{(k)}, \cdots, x_{n-1}^{(k)} \right\rangle \right\}_k$ in $P_0 \times \cdots \times P_{n-1}$ can be computed component-wise.

$$\bigsqcup_k^{\infty} \left\langle x_0^{(k)}, \cdots, x_{n-1}^{(k)} \right\rangle = \left\langle \bigsqcup_k^{\infty} x_0^{(k)}, \cdots, \bigsqcup_k^{\infty} x_{n-1}^{(k)} \right\rangle$$

(b) For every chain $\{z_k\}_k$ in $P_0 + \cdots + P_{n-1}$, there are some i and a chain $\{x_k\}_k$ in P_i such that $z_k = \langle i, x_k \rangle$ for all k.

(ii) These predomain constructors correspond to the sum and product type constructors in programming languages.

(iii) For each case, we have a way to construct an element and a way to destruct an element. Constructors:

(a) injection function $\iota_k \in \left[P_k \underset{c}{\to} P_0 + \cdots + P_{n-1} \right] : \iota_k(x) = \langle k, x \rangle$

(b) For $f_0 \in \left[P \underset{c}{\to} P_0 \right], \cdots, f_{n-1} \in \left[P \underset{c}{\to} P_{n-1} \right]$,
the "target-tupling" function $f_0 \otimes \cdots \otimes f_{n-1} \in \left[P \underset{c}{\to} P_0 \times \cdots \times P_{n-1} \right]$:

$$(f_0 \otimes \cdots \otimes f_{n-1})(x) = \left\langle f_0(x), \cdots, f_{n-1}(x) \right\rangle$$

Destructors:

(a) For $f_0 \in \left[P_0 \underset{c}{\to} P \right], \cdots, f_{n-1} \in \left[P_{n-1} \underset{c}{\to} P \right]$,
the "source-tupling" function $f_0 \oplus \cdots \oplus f_{n-1} \in \left[P_0 + \cdots + P_{n-1} \underset{c}{\to} P \right]$:

$$(f_0 \oplus \cdots \oplus f_{n-1}) \left(\langle i, x \rangle \right) = f_i(x)$$

(b) projection function $\pi_k \in \left[P_0 \times \cdots \times P_{n-1} \underset{c}{\to} P_k \right]$:

$$\pi_k \left(\langle x_0, \cdots, x_{n-1} \rangle \right) = x_k$$

These constructors and destructors are mutually inverse in a sense. Prop 5.1 and Prop 5.2 in the textbook express such inverse relationships.

(iv) The sum and product operators can be applied to continuous functions so as to build a new continuous function. For instance, if

$$f_0 \in \left[P_0 \underset{c}{\to} P_0' \right], \cdots, f_{n-1} \in \left[P_{n-1} \underset{c}{\to} P_{n-1}' \right]$$

Then we have the following two continuous functions:

$$\begin{aligned}
&(f_0 + \cdots + f_{n-1}) \in \left[P_0 + \cdots + P_{n-1} \underset{c}{\to} P_0' + \cdots + P_{n-1}' \right] \\
&(f_0 + \cdots + f_{n-1}) \langle i, x \rangle = \left\langle i, f_i(x) \right\rangle \\
&(f_0 \times \cdots \times f_{n-1}) \in \left[P_0 \times \cdots \times P_{n-1} \underset{c}{\to} P_0' \times \cdots \times P_{n-1}' \right] \\
&(f_0 \times \cdots \times f_{n-1}) \langle x_0, \cdots, x_{n-1} \rangle = \left\langle f_0(x_0), \cdots, f_{n-1}(x_{n-1}) \right\rangle
\end{aligned}$$

Many predomain constructors similarly induce constructors for continuous functions. This is because they are functors on the category of predomains and continuous functions. We will look at such category-theoretic formulation in some later lectures.

(6) One nontrivial important concept is a recursively-defined domain. Recall the description of Ω in the beginning of this lecture:

$$\begin{aligned}
\Omega &\simeq \left(\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to \Omega) \right)_\perp \\
\hat{\Sigma} &= \Sigma^{\clubsuit} + \Sigma^{\blacklozenge}
\end{aligned}$$

$\simeq$ means the presence of two continuous functions ϕ and ψ:

$$\Omega \underset{\psi}{\overset{\phi}{\rightleftarrows}} \left(\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to \Omega) \right)_\perp$$

such that $\phi \circ \psi = \mathrm{id}$ and $\psi \circ \phi = \mathrm{id}$.

Note that the RHS of $\simeq$ contains Ω itself, something that is being defined. It is a bit like Ω is defined in terms of itself, i.e. recursively. Or we can say that Ω is a fixed point of some equations over domains.

(i) Ω is the domain of possible outcomes. The isomorphism ϕ confirms that it consists of five kinds of elements corresponding to five different outcomes described earlier.

$^{\clubsuit}$normal termination
$^{\blacklozenge}$failed termination

(ii) Ω is not just a solution of the recursive domain equation. It satisfies the following minimality condition[♣].

For any domain D and any continuous function α

$$\alpha \in \left[\left(\hat{\Sigma} + (\mathbb{Z} \times D) + (\mathbb{Z} \to D) \right)_{\perp} \underset{c}{\to} D \right],$$

there exists a unique continuous function $\beta \in \left[\Omega \underset{c}{\to} D \right]$ such that the following diagram commutes.

$$\begin{array}{ccc} \left(\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to \Omega) \right)_{\perp} & \xrightarrow{\left(\mathrm{id}_{\hat{\Sigma}} + (\mathrm{id}_{\mathbb{Z}} \times \beta) + (\mathbb{Z} \to \beta)^{\blacklozenge} \right)} & \left(\hat{\Sigma} + (\mathbb{Z} \times D) + (\mathbb{Z} \to D) \right)_{\perp} \\ \Big\downarrow \phi & & \Big\downarrow \alpha \\ \Omega & \xrightarrow[\beta]{} & D \end{array}$$

One important consequence is that we can define a continuous function from Ω by case analysis, just as we can define a function on programs in a syntax-directed manner.

(iii) Why does such Ω exist? Because the predomain constructors used in the RHS of $\simeq$ are all very good, that is, they satisfy so-called local continuity. The situation is very similar to the one for the fixed point theorem. There we require a function to be continuous. There, we use an analogous property on an operator that constructs a domain. Later we will study this in detail.

[♣] also called initiality

[♦]
(a) $(\mathbb{Z} \to \beta) \in \left[(\mathbb{Z} \to \Omega) \underset{c}{\to} (\mathbb{Z} \to D) \right]$
$(\mathbb{Z} \to \beta)\,(g)\,(n) = \beta\left(g\,(n)\right)$

(b) $\forall f \in \left[D' \underset{c}{\to} D'' \right]$, we have $f_{\perp} \in \left[D'_{\perp} \underset{c}{\to} D''_{\perp} \right]$
s.t. $f_{\perp}\,(\perp) = \perp$ and $f_{\perp}\,(x) = f\,(x)$ for all $x \in D'$.

(iv) A few basic embedding operators[♣]:

$$\begin{aligned}
\iota_{\perp} &\in \left[\{\perp\} \underset{c}{\to} \Omega \right] & \iota_{\perp}(\perp) &= \psi(\perp_{\Omega}) \\
\iota_{\text{term}} &\in \left[\Sigma \underset{c}{\to} \Omega \right] & \iota_{\text{term}}(\sigma) &= \psi\left(\langle 0, \langle 0, \sigma\rangle\rangle\right) = \psi(\sigma)^{\blacklozenge} \\
\iota_{\text{abort}} &\in \left[\Sigma \underset{c}{\to} \Omega \right] & \iota_{\text{abort}}(\sigma) &= \psi\left(\langle 0, \langle 1, \sigma\rangle\rangle\right) = \psi\left(\langle \text{abort}, \sigma\rangle\right) \\
\iota_{\text{out}} &\in \left[\mathbb{Z} \times \Omega \underset{c}{\to} \Omega \right] & \iota_{\text{out}}(n, \omega) &= \psi\left(\langle 1, \langle n, \omega\rangle\rangle\right) = \psi\left(\langle n, \omega\rangle\right) \\
\iota_{\text{in}} &\in \left[(\mathbb{Z} \to \Omega) \underset{c}{\to} \Omega \right] & \iota_{\text{in}}(g) &= \psi\left(\langle 2, g\rangle\right) = \psi(g)
\end{aligned}$$

(v) Consider $f \in \left[\Sigma \underset{c}{\to} \Omega \right]$ and $h \in \left[\Sigma \underset{c}{\to} \Sigma \right]$. We want to define $f_* \in \left[\Omega \underset{c}{\to} \Omega \right]$ and $h_\dagger \in \left[\Omega \underset{c}{\to} \Omega \right]$ such that f_* applies f only for normally terminating state part of a given $\omega \in \Omega$ and $h_\dagger$ applies h applies h to the terminating or failing state part of ω. For instance,

$$\begin{aligned}
f_*\left(\iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, \iota_{\text{term}}(\sigma)\right)\right)\right) &= \iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, f(\sigma)\right)\right) \\
f_*\left(\iota_{\text{out}}\left(3, \iota_{\text{abort}}\sigma\right)\right) &= \iota_{\text{out}}\left(3, \iota_{\text{abort}}(\sigma)\right) \\
h_\dagger\left(\iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, \iota_{\text{term}}(\sigma)\right)\right)\right) &= \iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, \iota_{\text{term}}\left(h(\sigma)\right)\right)\right) \\
h_\dagger\left(\iota_{\text{out}}\left(3, \iota_{\text{abort}}\sigma\right)\right) &= \iota_{\text{out}}\left(3, \iota_{\text{abort}}\left(h(\sigma)\right)\right)
\end{aligned}$$

How to define f_* and $h_\dagger$? Because of (ii), we can define them by case analysis:

$$\begin{aligned}
f_*\left(\iota_{\perp}\right) &= \iota_{\perp}(\perp) \\
f_*\left(\iota_{\text{term}}(\sigma)\right) &= \iota_{\text{term}}\left(f(\sigma)\right) \\
f_*\left(\iota_{\text{abort}}(\sigma)\right) &= \iota_{\text{abort}}(\sigma) \\
f_*\left(\iota_{\text{out}}(n, \omega)\right) &= \iota_{\text{out}}\left(n, f_*(\omega)\right) \\
f_*\left(\iota_{\text{in}}(g)\right) &= \iota_{\text{in}}\left(\lambda n.\ f_*\left(g(n)\right)\right)
\end{aligned}$$

$$\begin{aligned}
h_\dagger\left(\iota_{\perp}\right) &= \iota_{\perp}(\perp) \\
h_\dagger\left(\iota_{\text{term}}(\sigma)\right) &= \iota_{\text{term}}\left(h(\sigma)\right) \\
h_\dagger\left(\iota_{\text{abort}}(\sigma)\right) &= \iota_{\text{abort}}\left(h(\sigma)\right) \\
h_\dagger\left(\iota_{\text{out}}(n, \omega)\right) &= \iota_{\text{out}}\left(n, h_\dagger(\omega)\right) \\
h_\dagger\left(\iota_{\text{in}}(g)\right) &= \iota_{\text{in}}\left(\lambda n.\ h_\dagger\left(g(n)\right)\right).
\end{aligned}$$

[♣]Editor's note: injection function?

[♦]We will just write σ. Similar for other cases.

Exercise 4.4. In both cases, we have well-defined f_* and $h_\dagger$ because of the minimality condition in (ii). Find an appropriate α and D.

(vi) For $\omega, \omega' \in \Omega$, intuitively $\omega \sqsubseteq \omega'$ if we can obtain ω' by replacing $\bot$ in ω. Intuitively, $\sqsubseteq$ represents the progression of computation.

Example. We will write $\bot_\Omega$ for $\iota_\bot(\bot)$.

$$\iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, \bot_\Omega\right)\right) \sqsubseteq \iota_{\text{out}}\left(3, \iota_{\text{out}}\left(4, \iota_{\text{out}}\left(5, \iota_{\text{term}}\left(\sigma\right)\right)\right)\right)$$

$$\iota_{\text{in}}\left(\lambda n.\ \iota_{\text{out}}\left(n+1, \bot_\Omega\right)\right) \sqsubseteq \iota_{\text{in}}\left(\lambda n.\ \iota_{\text{out}}\left(n+1, \iota_{\text{in}}\left(\lambda m.\ \iota_{\text{out}}\left(m+n, \iota_{\text{term}}\left(\sigma\right)\right)\right)\right)\right)$$

(7) Interpretation of commands:

$$\begin{aligned}
[\![-]\!] &\in \left[\langle comm\rangle \to \left[\Sigma \underset{c}{\to} \Omega\right]\right] \\
[\![\text{skip}]\!]\left(\sigma\right) &= \iota_{\text{term}}\left(\sigma\right) \\
[\![c_1;\ c_2]\!]\left(\sigma\right) &= [\![c_2]\!]_*\left([\![c_1]\!]\left(\sigma\right)\right) \\
[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!]\left(\sigma\right) &= \text{if } [\![b]\!]\left(\sigma\right) \text{ then } [\![c_1]\!]\left(\sigma\right) \text{ else } [\![c_2]\!]\left(\sigma\right) \\
[\![\text{while } b \text{ do } c]\!]\left(\sigma\right) &= \left(Y_{\underset{c}{(\Sigma \to \Omega)}} F\right)^{\clubsuit}\left(\sigma\right) \\
\text{where } F\left(f\right)\left(\sigma\right) &= \text{if } [\![b]\!]\left(\sigma\right) \text{ then } f_*\left([\![c]\!]\left(\sigma\right)\right) \text{ else } \sigma \\
[\![\text{fail}]\!]\left(\sigma\right) &= \iota_{\text{abort}}\left(\sigma\right) \\
[\![!e]\!]\left(\sigma\right) &= \iota_{\text{out}}\left([\![e]\!]\left(\sigma\right), \iota_{\text{term}}(\sigma)\right) \\
[\![?v]\!]\left(\sigma\right) &= \iota_{\text{in}}\left(\lambda n.\ \iota_{\text{term}}\left(\left[\sigma \mid v : n\right]\right)\right) \\
[\![\text{newvar } v := e \text{ in } c]\!]\left(\sigma\right) &= \left(\lambda\sigma'.\ \left[\sigma' \mid v : \sigma\left(v\right)\right]\right)_\dagger\left([\![c]\!]\left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right) \\
[\![v := e]\!]\left(\sigma\right) &= \iota_{\text{term}}\left(\left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right)
\end{aligned}$$

Exercise 4.5. Prove the following equations:

$$[\![x := 3;\ !x]\!] = [\![x := 3;\ !3]\!]$$
$$[\![\text{fail};\ c]\!] = [\![\text{fail}]\!]$$

Exercise 4.6. Does the following equation hold?

$$[\![?x;\ y := 3]\!] = [\![y := 3;\ ?x]\!]$$

$^{\clubsuit} \bigsqcup_{n=0}^{\infty} F^n\left(\bot\right)$

4.4 Continuation Semantics

(1) The continuation semantics is an alternative, more general way of interpreting commands. The key concept here is continuation, which is some mathematical entity representing the rest of computation. Intuitively, a continuation denotes what will happen all the way until the end when the current command finishes normally. In the continuation semantics, we interpret a command as a function that takes a continuation κ and a state σ, and computes (or returns) the ultimate answer of the entire computation.

(2) Let's ignore the command "fail" for now. Mathematically, the continuation semantics defines the following function:

$$[\![-]\!]^{\text{cont}} \in \left[\langle comm \rangle \to \left(\Sigma \underset{c}{\to} \Omega \right) \underset{c}{\to} \Sigma \underset{c}{\to} \Omega \right]$$

Compare it with the (direct) semantics that we looked at earlier:

$$[\![-]\!] \in \left[\langle comm \rangle \to \Sigma \underset{c}{\to} \Omega \right]$$

We have the extra part "$\left(\Sigma \underset{c}{\to} \Omega \right) \underset{c}{\to} \Sigma$", which expresses that $[\![-]\!]^{\text{cont}}$ takes an additional continuation parameter κ, compared with the (direct) semantics $[\![-]\!]$.

Thus, when $\kappa \in \left[\Sigma \underset{c}{\to} \Omega \right]$ and $\sigma \in \Sigma$,

$$\omega^{\clubsuit} = [\![c]\!]^{\text{cont}} \left(\kappa^{\blacklozenge}\right) \left(\sigma^{\blacktriangledown}\right) \in \Omega$$

(3) A good way to learn this continuation semantics is to complete the definition of $[\![-]\!]^{\text{cont}}$ based on the intuition that we discussed so far. So, hide the semantic clause

$^{\clubsuit}$result of running c and then κ at the state σ
$^{\blacklozenge}$computation to be done after c
$^{\blacktriangledown}$initial state

in each case, and try to rediscover the clause for yourself.

$$[\![c]\!]^{\text{cont}} \in \left[\left[\Sigma \underset{c}{\to} \Omega \right] \underset{c}{\to} \Sigma \underset{c}{\to} \Omega \right]$$

$$[\![\text{skip}]\!]^{\text{cont}}\, \kappa\, \sigma = \kappa\,(\sigma)$$

$$[\![c_0;\ c_1]\!]^{\text{cont}}\, \kappa\, \sigma = [\![c_0]\!]^{\text{cont}}\, \left(\lambda\sigma'.\ [\![c_1]\!]^{\text{cont}}\, \kappa\, \sigma'\right) \sigma$$

$$[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!]_{\text{cont}}\, \kappa\, \sigma = \text{if } [\![b]\!]\,(\sigma) \text{ then } [\![c_0]\!]^{\text{cont}}\, \kappa\, \sigma \text{ else } [\![c_1]\!]^{\text{cont}}\, \kappa\, \sigma$$

$$[\![\text{while } b \text{ do } c_0]\!]_{\text{cont}}\, \kappa\, \sigma = \left(Y_{\underset{c}{\Sigma \to \Omega}}\,(F) \right)(\sigma) = \left(\bigsqcup_{n=0}^{\infty} F^n\,(\bot) \right)(\sigma)$$

$$\text{where } F \in \left(\left[\Sigma \underset{c}{\to} \Omega \right] \underset{c}{\to} \left[\Sigma \underset{c}{\to} \Omega \right] \right)$$

$$F\left(\kappa'\right)\left(\sigma'\right) = \text{if } [\![b]\!]\left(\sigma'\right) \text{ then } [\![c_0]\!]^{\text{cont}}\, \kappa'\sigma' \text{ else } \kappa\left(\sigma'\right)$$

As usual, the most tricky part is the case for while. Note that we define the operator F on the domain of continuation, $\left[\Sigma \underset{c}{\to} \Omega \right]$, not on the domain of commands $\left[\left[\Sigma \underset{c}{\to} \Omega \right] \underset{c}{\to} \Sigma \underset{c}{\to} \Omega \right]$, and then we use the least fixed point of F, which denotes some continuation in $\left[\Sigma \underset{c}{\to} \Omega \right]$. In the (direct) semantics, we used the domain of commands.

(4) Our definition is not complete. We should define $[\![?v]\!]^{\text{cont}}$, $[\![!e]\!]^{\text{cont}}$ and $[\![\text{newvar } v := e \text{ in } c]\!]^{\text{cont}}$.

$$[\![?v]\!]^{\text{cont}}\, \kappa\, \sigma = \iota_{\text{in}} \left(\lambda n \in \mathbb{Z}.\ \kappa\left([\sigma \mid v : n] \right) \right)^{\clubsuit}$$

$$[\![!e]\!]^{\text{cont}}\, \kappa\, \sigma = \iota_{\text{out}} \left([\![e]\!]\, \sigma, \kappa\,(\sigma) \right)^{\blacklozenge}$$

$$[\![\text{newvar } v := e \text{ in } c]\!]^{\text{cont}}\, \kappa\, \sigma = [\![c]\!]^{\text{cont}} \left(\lambda\sigma'.\ \kappa\left(\left[\sigma' \;\middle|\; v : \sigma\,(v) \right] \right) \right) \left[\sigma \;\middle|\; v : [\![e]\!]\,\sigma \right]$$

(5) The (direct) semantics and the continuation semantics are closely related. In a sense, this should be the case because both semantics are dealing with the same thing. Here is the formal relationship:

$$[\![c]\!]^{\text{cont}}\, \kappa\, \sigma = \kappa_*\left([\![c]\!]\,(\sigma) \right)$$

Exercise 4.7. Prove the above equation.

$^{\clubsuit}$recall $\iota_{\text{out}} \in \left[\mathbb{Z} \times \Omega \underset{c}{\to} \Omega \right]$

$^{\blacklozenge}$recall $\iota_{\text{in}} \in \left[\left[\mathbb{Z} \underset{c}{\to} \Omega \right] \underset{c}{\to} \Omega \right]$

(6) So far we didn't consider the command fail. Now it is time to stop ignoring it. Can we add a semantic clause for $[\![\text{fail}]\!]^{\text{cont}}$? What about the following?

$$[\![\text{fail}]\!]^{\text{cont}}\ \kappa\ \sigma = \iota_{\text{abort}}\left(\sigma\right)$$

Unfortunately, this simple definition doesn't work:

$$\begin{aligned}
&[\![\text{newvar } x := 1 \text{ in fail}]\!]^{\text{cont}}\ \kappa\ \sigma \quad \text{where } \sigma\left(x\right) = 0\\
&= [\![\text{fail}]\!]^{\text{cont}} \left(\lambda\sigma'.\ \kappa\left(\left[\sigma' \mid x : \sigma\left(x\right)\right]\right)\right) \left[\sigma \mid x : [\![1]\!]\,\sigma\right]\\
&= \iota_{\text{abort}}\left(\left[\sigma \mid x : 1\right]\right)\\
&\neq \iota_{\text{abort}}\left(\sigma\right)
\end{aligned}$$

(7) One solution is to pass two continuations. One for normal computation and the other for aborted computation.

$$[\![-]\!]_2^{\text{cont}} \in \left[\langle comm\rangle \to \left[\Sigma \underset{c}{\to} \Omega\right]^{\clubsuit} \underset{c}{\to} \left[\Sigma \underset{c}{\to} \Omega\right]^{\blacklozenge} \underset{c}{\to} \Sigma \underset{c}{\to} \Omega\right]$$

$$\begin{aligned}
[\![v := e]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \kappa_t\left(\left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right)\\
[\![\text{skip}]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \kappa_t\left(\sigma\right)\\
[\![c_1;\ c_2]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= [\![c_1]\!]_2^{\text{cont}}\left([\![c_2]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\right)\kappa_f\ \sigma\\
[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \text{if } [\![b]\!]\,\sigma \text{ then } [\![c_0]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma \text{ else } [\![c_1]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma\\
[\![\text{while } b \text{ do } c]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \left(Y_{\underset{c}{\Sigma\to\Omega}}\left(F\right)\right)\left(\sigma\right) = \left(\bigsqcup_{n=0}^{\infty} F^n\left(\bot\right)\right)\left(\sigma\right)\\
\text{where } F\left(\kappa\right)\left(\sigma'\right) &= \text{if } [\![b]\!]\,\sigma' \text{ then } [\![c]\!]_2^{\text{cont}}\ \kappa\ \kappa_f\ \sigma' \text{ else } \kappa_t\left(\sigma'\right)\\
[\![\text{newvar } v := e \text{ in } c]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= [\![c]\!]_2^{\text{cont}}\ \kappa_t'\ \kappa_f'\ \left[\sigma \mid v : [\![e]\!]\,\sigma\right]\\
\text{where } \kappa_t' &= \left(\lambda\sigma'.\ \kappa_t\left(\left[\sigma' \mid v : \sigma\left(v\right)\right]\right)\right)\\
\text{and } \kappa_f' &= \left(\lambda\sigma'.\ \kappa_f\left(\left[\sigma' \mid v : \sigma\left(v\right)\right]\right)\right)\\
[\![\text{fail}]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \kappa_f\left(\sigma\right)\\
[\![!e]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \iota_{\text{out}}\left([\![e]\!]\,\sigma, \kappa_t\left(\sigma\right)\right)\\
[\![?v]\!]_2^{\text{cont}}\ \kappa_t\ \kappa_f\ \sigma &= \iota_{\text{in}}\left(\lambda n \in \mathbb{Z}.\ \kappa_t\left(\left[\sigma \mid v : n\right]\right)\right)
\end{aligned}$$

Note that κ_f hardly changes. The only change is to make it restore locally declared variables to their original values.

$^{\clubsuit}$for normal computation
$^{\blacklozenge}$for aborted computation

Chapter 5

Transition Semantics

5.1 Motivation or objective

(1) So far we defined the meanings of programs in imperative languages using the denotational semantics. A good denotational semantics reveals an underlying mathematical structure of a programming language and hides the intermediate steps of computation as much as possible. Also, it is compositional, and lets us reason about a piece of program code even when we do not know its surrounding program context.

(2) However, when a programming language has advanced or complex language constructs, defining a denotational semantics of the language may be difficult. Also, sometimes we want to have a mathematical semantics of programs that tells us what happens in the middle of computation.

(3) The operational semantics is an alternative approach to give mathematical meanings to programs. It is non-compositional, and does not hide the intermediate steps of computation. But it is usually very simple and also rigorous or formal enough to enable a mathematical study of a programming language and language tools such as compiler and program verifier. Also, an operational semantics of a programming language often serves as a blueprint of an interpreter or a compiler of the language.

(4) In this chapter, we will study the so-called small-step operational semantics, which Reynolds calls transition semantics.

5.2 Main idea of the small-step operational semantics

(1) The key idea is to formalize one computation step of a program using a relation, called transition relation.

(2) Typically, a small-step operational semantics has two main parts.

(i) Γ . . . a set of configurations.
Usually, $\Gamma = \Gamma_N \cup \Gamma_T$ for some sets Γ_N, Γ_T with $\Gamma_N \cap \Gamma_T = \emptyset$. Each element $\gamma \in \Gamma$ describes the status of a machine that runs a program. If $\gamma \in \Gamma_N$,

it is called nonterminal configuration and the execution of its program is not finished yet. If $\gamma \in \Gamma_T$, it is called terminal configuration and the execution of its program is finished.

(ii) $\rightarrow \subseteq \Gamma_N \times \Gamma$ ... transition relation.
Intuitively, $(\gamma, \gamma') \in \rightarrow$ [♣] means that one computation step changes the status of a machine from γ to γ'. Note that γ has to be a nonterminal configuration because of the domain of $\rightarrow$. This condition is consistent with the intuition behind nonterminal and terminal configurations.

Defining a small-step operational semantics amounts to defining Γ, Γ_N, Γ_T, and $\rightarrow$. We will see a few examples of the operational semantics in this lecture. Often if we define Γ, Γ_N, Γ_T, then the definition of $\rightarrow$ follows almost automatically. This is a bit similar to the situation in the denotational semantics that if the form of the interpretation function for commands $[\![-]\!]$ is determined, the actual definition of the function follows almost automatically.

(3) When defining the $\rightarrow$ relation, we usually use the inference rule notation $\dfrac{\varphi_1 \ \cdots \ \varphi_n}{\varphi}$ that you saw when we discussed Hoare logic.

5.3 Small-step operational semantics of the simple imperative language

(1) Let's try to give the operational semantics to the simple imperative language that we studied. Here is a reminder of this abstract grammar:

$\langle comm \rangle$ ::= skip
| $\langle var \rangle := \langle intexp \rangle$
| $\langle comm \rangle$; $\langle comm \rangle$
| if $\langle boolexp \rangle$ then $\langle comm \rangle$ else $\langle comm \rangle$
| while $\langle boolexp \rangle$ do $\langle comm \rangle$
| while $\langle boolexp \rangle$ do $\langle comm \rangle$

(2) What should we do? First, we have to define the set of nonterminal configurations [♦] and that of terminal configurations. [♥]

Here are our definitions:

$$\Gamma_N \stackrel{\text{def}}{=} \langle comm \rangle \text{[♠]} \times \Sigma^{\star} \qquad \Gamma_T \stackrel{\text{def}}{=} \Sigma^{\text{[❀]}}$$

The set of configurations is the union of the above two sets.

[♣] typically written as $\gamma \rightarrow \gamma'$
[♦] Γ_N
[♥] Γ_T
[♠] command that records the remaining computation
[⋆] the current state of a machine
[❀] the $\langle comm \rangle$ part is missing because there is no remaining computation

(3) Second, we should define a binary relation

$$\rightarrow \subseteq \Gamma_N \times \Gamma,$$

called transition relation, that describes single-step computation. We write $\big(\gamma, \gamma'\big)$ to mean $\big(\gamma, \gamma'\big) \in \rightarrow$. We define the transition relation $\rightarrow$ using the inference rule notation.

$$\frac{}{\langle \text{skip}, \sigma\rangle \rightarrow \sigma} \qquad \frac{}{\langle v := e, \sigma\rangle \rightarrow \sigma\big(v \mapsto [\![e]\!]\,\sigma\big)}$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \sigma'}{\langle c_1;\ c_2, \sigma\rangle \rightarrow \big\langle c_2, \sigma'\big\rangle} \qquad \frac{\langle c_1, \sigma\rangle \rightarrow \big\langle c_1', \sigma'\big\rangle}{\langle c_1;\ c_2, \sigma\rangle \rightarrow \big\langle c_1';\ c_2, \sigma'\big\rangle}$$

$$\frac{}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}\ \big([\![b]\!]\,\sigma = \mathbf{tt}\big)$$

$$\frac{}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle}\ \big([\![b]\!]\,\sigma = \mathbf{ff}\big)$$

$$\frac{}{\langle \text{while } b \text{ do } c, \sigma\rangle \rightarrow \sigma}\ \big([\![b]\!]\,\sigma = \mathbf{ff}\big)$$

$$\frac{}{\langle \text{while } b \text{ do } c, \sigma\rangle \rightarrow \langle c;\ \text{while } b \text{ do } c, \sigma\rangle}\ \big([\![b]\!]\,\sigma = \mathbf{tt}\big)$$

Note that the right-hand side of $\rightarrow$ may include a command that is not a sub-command of the one on the left-hand side. Look at $\langle c_1;\ c_2, \sigma\rangle \rightarrow \big\langle c_1';\ c_2, \sigma'\big\rangle$ and $\langle \text{while } b \text{ do } c, \sigma\rangle \rightarrow \langle c;\ \text{while } b \text{ do } c, \sigma\rangle$. This indicates that the semantics is not compositional. All these rules correspond to our intuitive understading of one computation step. They can form the basis of the implementation of a simple interpreter, which just needs to run the $\rightarrow$ step repeatedly.

(4) Formal properties of the operational semantics:

(i) $\gamma \rightarrow \gamma_1$ and $\gamma \rightarrow \gamma_2 \quad \Longrightarrow \quad \gamma_1 = \gamma_2$.
The semantics is deterministic.

(ii) $\forall \gamma \in \Gamma_N\ \exists \gamma'$ s.t. $\gamma \rightarrow \gamma'$.
In this semantics, executions never get stuck.

(iii) From (i) to (ii), it follows that for every $\gamma \in \Gamma$, there exists a unique maximal sequence (may be infinite)

$$\gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_n$$

such that

$$\gamma = \gamma_0 \wedge \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_n$$
$$\wedge\ \gamma_n \text{ is a terminal configuration or } n \text{ is infinite.}$$

This maximal finite or infinite sequence represents the full computation starting from γ.

(iv) We write $\gamma \uparrow$ if the maximal execution sequence from γ is infinite. Then, for all commands c and states σ,

$$\begin{aligned} [\![c]\!]\,\sigma = \bot \quad &\text{iff} \quad \langle c, \sigma\rangle \uparrow \\ [\![c]\!]\,\sigma = \sigma' \quad &\text{iff} \quad \langle c, \sigma\rangle \to^{*\,\clubsuit} \sigma' \end{aligned}$$

Exercise 5.1. Prove (i), (ii), and (iv).

Exercise 5.2. Explain why the reasoning in (iii) is true.

5.4 Extension with newvar

(1) Extend the language with variable declaration:

$$\langle comm\rangle ::= \ \ldots \ \ | \text{ newvar } \langle var\rangle := \langle intexp\rangle \text{ in } \langle comm\rangle$$

(2) How should we modify the $\to$ relation? Add a rule for newvar:

(i) Option 1:

$$\frac{}{\langle \text{newvar } v := e \text{ in } c, \sigma\rangle \to \left\langle c;\ v := n, \left[\sigma \;\middle|\; v \mapsto [\![e]\!]\,\sigma\right]\right\rangle}$$

(ii) Option 2

$$\frac{\left\langle c, \left[\sigma \;\middle|\; v : [\![e]\!]\,\sigma\right]\right\rangle \to \sigma'}{\langle \text{newvar } v := e \text{ in } c, \sigma\rangle \to \left[\sigma' \;\middle|\; v : [\![e]\!]\,\sigma\right]}$$

(iii) Both options are acceptable. But option 2 is better. Only option 2 works when we extend the language with primitives for concurrent executions.

(3) Note that we did not change Γ_N and Γ_T. Thus, adding newvar doesn't change the operational semantics much. In a sense, this small change means that newvar doesn't change the language much, either.

5.5 Adding fail

$$\langle comm\rangle ::= \ \ldots \ \ | \text{ fail}$$

(1) When we add fail, we have to change the set Γ_T of terminal configuration, because we now have two types of terminations, normal one and abnormal one.

$$\Gamma_T \stackrel{\text{def}}{=} \Sigma \cup \{\text{abort}\} \times \Sigma \quad (\text{or } \ = \Sigma + \Sigma)$$

Γ_N remains unchanged.

$^{\clubsuit}$reflexive and transitive closure of $\to$, i.e. $\to^* \stackrel{\text{def}}{=} \bigcup_n^\infty (\to)^n$

(2) Since Γ_T and so Γ are changed, we should change the definition of $\rightarrow$. We will also have to add a rule for fail. Here is the new set of rules.

$$\frac{}{\langle \text{fail}, \sigma\rangle \rightarrow \langle \text{abort}, \sigma\rangle}\,\clubsuit \qquad \frac{}{\langle \text{skip}, \sigma\rangle \rightarrow \sigma} \qquad \frac{}{\langle v := e, \sigma\rangle \rightarrow \sigma\left(v \mapsto [\![e]\!]\,\sigma\right)}$$

$$\frac{}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}\left([\![b]\!]\,\sigma = \mathbf{tt}\right)$$

$$\frac{}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle}\left([\![b]\!]\,\sigma = \mathbf{ff}\right)$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \left\langle c_1', \sigma'\right\rangle}{\langle c_1;\ c_2, \sigma\rangle \rightarrow \left\langle c_1';\ c_2, \sigma'\right\rangle} \qquad \frac{\langle c_1, \sigma\rangle \rightarrow \sigma'}{\langle c_1;\ c_2, \sigma\rangle \rightarrow \left\langle c_2, \sigma'\right\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \left\langle \text{abort}, \sigma'\right\rangle}{\langle c_1;\ c_2, \sigma\rangle \rightarrow \left\langle \text{abort}, \sigma'\right\rangle}\,\blacklozenge$$

$$\frac{}{\langle \text{while } b \text{ do } c, \sigma\rangle \rightarrow \sigma}\left([\![b]\!]\,\sigma = \mathbf{ff}\right)$$

$$\frac{\langle c, \sigma\rangle \rightarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma\rangle \rightarrow \left\langle c;\ \text{while } b \text{ do } c, \sigma'\right\rangle}\left([\![b]\!]\,\sigma = \mathbf{tt}\right)$$

$$\frac{\left\langle c, \left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right\rangle \rightarrow \left\langle \text{abort}, \sigma'\right\rangle}{\langle \text{newvar } v := e \text{ in } c, \sigma\rangle \rightarrow \left\langle \text{abort}, \left[\sigma' \mid v : \sigma\,(v)\right]\right\rangle}\,\blacktriangledown$$

$$\frac{\left\langle c, \left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right\rangle \rightarrow \sigma'}{\langle \text{newvar } v := e \text{ in } c, \sigma\rangle \rightarrow \left[\sigma' \mid v : \sigma\,(v)\right]}$$

$$\frac{\left\langle c, \left[\sigma \mid v : [\![e]\!]\,\sigma\right]\right\rangle \rightarrow \left\langle c', \sigma'\right\rangle}{\langle \text{newvar } v := e \text{ in } c, \sigma\rangle \rightarrow \left\langle \text{newvar } v := \sigma'\,(v)\ \text{ in } c', \left[\sigma' \mid v : \sigma\,(v)\right]\right\rangle}$$

$\clubsuit$ new rule: due to fail
$\blacklozenge$ new rule: due to the change of Γ_T
$\blacktriangledown$ new rule: due to the change of Γ_T

5.6 Handling input and output

$$\langle comm \rangle ::= \ \ldots \ \mid !\langle var \rangle \mid ?\langle var \rangle$$

(1) This time we have to change the form or type of $\to$. It is no longer a binary relation, but a ternary relation.

$$\to \subseteq \Gamma_N \times \Lambda \times \Gamma$$

$$\lambda \in \Lambda \stackrel{\text{def}}{=} \{\varepsilon\}^{\clubsuit} \cup \{?n \mid n \in \mathbb{Z}\}^{\blacklozenge} \cup \{!n \mid n \in \mathbb{Z}\}^{\blacktriangledown}$$

We write $\langle c, \sigma \rangle \xrightarrow{\lambda} \gamma$ to mean $\left\langle \langle c, \sigma \rangle, \lambda, \gamma \right\rangle \in \to$. We also often omit λ if $\lambda = \varepsilon$.

(2) Why do we make this change? It is because adding $?n$ and $!n$ to the language makes it necessary to describe some aspects of intermediate steps of computations explicitly.

(3) We include all the rules that (except the ones for $c_1;\ c_2$ and newvar) that we defined in §5.5. Of course, the occurrences of $\to$ in those old rules should be understood as $\xrightarrow{\varepsilon}$ with ε omitted for simplicity. In addition to these rules, we have the following rules:

$$\frac{}{\langle ?v, \sigma \rangle \xrightarrow{?n} \left[\sigma \mid v : \sigma(n)\right]} \qquad \frac{}{\langle !e, \sigma \rangle \xrightarrow{!\llbracket e \rrbracket \sigma} \sigma}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \sigma'}{\langle c_0;\ c_1, \sigma \rangle \xrightarrow{\lambda} \langle c_1, \sigma' \rangle} \qquad \frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \langle c_0', \sigma' \rangle}{\langle c_0;\ c_1, \sigma \rangle \xrightarrow{\lambda} \langle c_0';\ c_1, \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \langle \text{abort}, \sigma' \rangle}{\langle c_0;\ c_1, \sigma \rangle \xrightarrow{\lambda} \langle \text{abort}, \sigma' \rangle}$$

$$\frac{\left\langle c, \left[\sigma \mid v : \llbracket e \rrbracket \sigma\right] \right\rangle \xrightarrow{\lambda} \sigma'}{\langle \text{newvar } v := e \text{ in } c, \sigma \rangle \xrightarrow{\lambda} \left[\sigma' \mid v : \sigma(v)\right]}$$

$$\frac{\left\langle c, \left[\sigma \mid v : \llbracket e \rrbracket \sigma\right] \right\rangle \xrightarrow{\lambda} \langle \text{abort}, \sigma' \rangle}{\langle \text{newvar } v := e \text{ in } c, \sigma \rangle \xrightarrow{\lambda} \left\langle \text{abort}, \left[\sigma' \mid v : \sigma(v)\right] \right\rangle}$$

$$\frac{\left\langle c, \left[\sigma \mid v : \llbracket e \rrbracket \sigma\right] \right\rangle \xrightarrow{\lambda} \langle c', \sigma' \rangle}{\langle \text{newvar } v := e \text{ in } c, \sigma \rangle \xrightarrow{\lambda} \left\langle \text{newvar } v := \sigma'(v) \text{ in } c', \left[\sigma' \mid v : \sigma(v)\right] \right\rangle}$$

Whenever an old rule contains a premise, we copy the rule and put λ above $\to$ in the premise and the conclusion, so that the label λ gets propagated from the execution of a subcommand to that of the original command.

$^{\clubsuit}$transition or execution without input or output
$^{\blacklozenge}$transition with an input
$^{\blacktriangledown}$transition with an output

(4) This operational semantics corresponds to the denotational semantics that we studied. The correspondence is formalized by the function F in page 134 of the textbook:

$$F(\gamma) = \begin{cases} \bot & \text{when } \gamma \uparrow \\ \iota_{\text{term}}\left(\sigma'\right) & \text{when } \gamma \to^* \sigma' \\ \iota_{\text{abort}}\left(\sigma'\right) & \text{when } \gamma \to^* (\mathit{abort}, \sigma') \\ \iota_{\text{out}}\left(n, F_{\gamma''}\right) & \text{when } \exists \gamma' \in \Gamma.\ \gamma \to^* \gamma' \text{ and } \gamma' \xrightarrow{!n} \gamma'' \\ \iota_{\text{in}}\left(\lambda n \in \mathbb{Z}.F_{\gamma_n}\right) & \text{when } \exists \gamma' \in \Gamma. \forall n \in \mathbb{Z}. \gamma \to^* \gamma' \text{ and } \gamma' \xrightarrow{?n} \gamma_n. \end{cases}$$

Intuitively, F runs a configuration until it finishes, or outputs a number, or waits for an input. F then returns what it gets when it completes this execution.

In a sense, the correspondence says that the denotational semantics comes from the operational semantics after unobservable intermediate states are abstracted away. For detail, look at the textbook.

Chapter 6

An Introduction to Category Theory

Chapter 8 of Tennent's book

6.1 Motivation

(1) The category theory is a branch of mathematics that studies essentially same or common notions and principles in different areas of mathematics. It is very abstract, but provides good guidelines about finding right definitions and right or meaningful questions to ask in a new mathematical theory.

(2) The category theory had huge influence on the research on programming languages, such as Scala, Haskell, and Rust.

(3) In this course, we will focus on the influence of the category theory on the semantics research. We will study abstract categorical concepts that give rise to popular notions appearing in the semantics of programming languages. Another big objective is to understand a result in the category theory (or a categorical formulation of domain theory) that ensures the existence of certain recursively defined domains, such as the following Ω that you saw in the chapter 5 of Reynolds' book:

$$\Omega \simeq \left(\hat{\Sigma} + \mathbb{Z} \times \Omega + [\,\mathbb{Z} \to \Omega\,]\right)_\perp$$
$$\hat{\Sigma} \stackrel{\text{def}}{=} \Sigma + \Sigma$$

(4) We will study only a tiny part of the category theory. If you are excited about it and are willing to read a math book, I recommend Mac Lane's "Categories for the Working Mathematician".

6.2 Definition of Category

Definition. A category $\mathcal{C}$ is a tuple $\big(\mathrm{Obj}, \mathrm{Hom}, \circ, \mathrm{id}\big)$ where

(1) Obj is a collection of elements called objects;

(2) for all objects $x, y \in \mathrm{Obj}$, $\mathrm{Hom}[x, y]$ is a collection of elements called morphisms from x to y;

(3) for all objects $x, y, z \in \mathrm{Obj}$, $\circ_{x,y,z}$ (or simply $\circ$) is a map from $\mathrm{Hom}[y, z] \times \mathrm{Hom}[x, y]$ to $\mathrm{Hom}[x, z]$, and is called composition;

(4) for every object $x \in \mathrm{Obj}$, id_x is an element in $\mathrm{Hom}[x, x]$, and is called identity morphism;

(5) these data should satisfy associativity and identity axioms:

(a) Associativity:

$$\begin{aligned}
&\forall w, x, y, z \in \mathrm{Obj}, \\
&\forall f \in \mathrm{Hom}[w, x], \\
&\forall g \in \mathrm{Hom}[x, y], \\
&\forall h \in \mathrm{Hom}[y, z], \\
&h \circ (g \circ f) = (h \circ g) \circ f
\end{aligned}$$

(b) Identity:

$$\begin{aligned}
&\forall x, y \in \mathrm{Obj}, \\
&\forall f \in \mathrm{Hom}[x, y], \\
&f \circ \mathrm{id}_x = f = \mathrm{id}_y \circ f = f
\end{aligned}$$

(1) Although not perfect, a reasonably good intuition is that a category $\mathcal{C}$ is a collection of spaces. ♣ The Obj part of $\mathcal{C}$ consists of spaces, and the $\mathrm{Hom}[x, y]$ part of $\mathcal{C}$ consists of maps between spaces that preserve the structures of the spaces. $\circ$ is then the usual function composition and id_x is the identity function on x.

(2) Here are some examples that match the intuition that I just explained:

(i) Set ... category of all sets

- Obj is the collection of all sets
- $\mathrm{Hom}[x, y]$ is the set of all functions from x to y. Note that since $x, y \in \mathrm{Obj}$, they are sets.
- $\circ$ is usual function composition
- id_x is the identity function on x

(ii) Predom ... category of all predomains and continuous functions

- Obj is the collection of all predomains
- $\mathrm{Hom}[x, y]$ is the set of all continuous functions from x to y.
- $\circ$ is the function composition
- id_x is the identity function on x

(iii) Dom ... Category of domains and continuous functions

- Obj is the collection of all domains
- $\mathrm{Hom}[x, y]$ is the set of all continuous functions from x to y.

♣ spaces that you encounter in mathematics, such as metric spaces, vector spacess, topological spaces, etc.

- $\circ$ is the function composition
- id_x is the identity function on x

Note that Dom is in a sense included in Predom. (Technically, it is a full subcategory of Predom.)

(3) As indicated by my use of the phrase "not perfect", there are categories that do not match the intuition well. Here is a very well-known example:

Let $(X, \sqsubseteq)$ be a partially ordered set. It can be understood as a category:

- $\mathrm{Obj} = X$
- $\mathrm{Hom}\,[\,x, y\,] = \begin{cases} \emptyset & \text{if } x \not\sqsubseteq y \\ \{\star\}^{\clubsuit} & \text{if } x \sqsubseteq y \end{cases}$
- $\mathrm{id}_X = \star$
- $\circ_{x,y,z} \in \big[\,\mathrm{Hom}\,[\,y, z\,] \times \mathrm{Hom}\,[\,x, y\,] \mapsto \mathrm{Hom}\,[\,x, z\,]\,\big]$
  If $x \sqsubseteq y$ and $y \sqsubseteq z$ (i.e., $\mathrm{Hom}\,[\,x, y\,] \neq \emptyset$ and $\mathrm{Hom}\,[\,y, z\,] \neq \emptyset$), then $\circ_{x,y,z}$ is the constant function to the unique element in $\mathrm{Hom}\,[\,x, z\,]$. ($\mathrm{Hom}\,[\,x, z\,] \neq \emptyset$ in this case because of the transitivity of $\sqsubseteq$.)
  Otherwise (i.e., $\mathrm{Hom}\,[\,x, y\,] = \emptyset$ or $\mathrm{Hom}\,[\,y, z\,] = \emptyset$), $\circ_{x,y,z}$ is the empty function (the function whose graph is the empty set).

(4) In the category theory, we often use commutative diagrams to express the equality of two morphisms. For instance,

$$\begin{array}{ccc} x & \xrightarrow{f} & y \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle g} \\ a & \xrightarrow{h} & z \end{array}$$

expresses that x, y, a, z are objects in a category and f, g, h, k are morphisms with domains and codomains indicated by the arrows (for instance, $f \in \mathrm{Hom}\,[\,x, y\,]$), and $g \circ f = h \circ k$. $^{\blacklozenge}$

(5) Another intuition about categories is that objects in a category are types in a programming language and morphisms from x to y are functions in the language from the input of type x to the output of type y.

6.3 Initial and terminal objects. Product and co-product

(1) In a category $\mathcal{C}$, we can build a new object out of existing ones. Often this new object satisfies one of the well-known conditions, and has a well-known name. We will study a few such names.

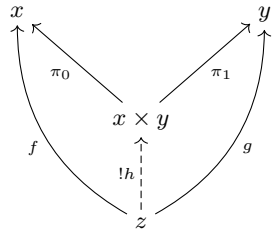
$^{\clubsuit}$a set with one element. It doesn't matter what that element is.
$^{\blacklozenge}$the most important bit

(2) Consider a category $\mathcal{C}$ and its objects $x, y \in \mathrm{Obj}$.

(i) An object z is a product of x and y, often written as $z = x \times y$, if there are morphisms $\pi_0 \in \mathrm{Hom}\,[z, x]$[♣] and $\pi_1 \in \mathrm{Hom}\,[z, y]$[♦] s.t.

for all objects w and morphisms $f : w \to x$[♥] and $g : w \to y$, there exists a unique morphism $h : w \to x \times y$[♠] with $f = \pi_0 \circ h$ and $g = \pi_1 \circ h$.

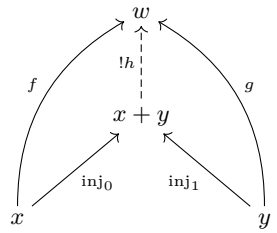
In practice,[★]



(ii) An object z is a co-product (or sum) of x and y, often written as $z = x + y$, if there are morphisms $\mathrm{inj}_0 : x \to x + y$ and $\mathrm{inj}_1 : y \to x + y$ s.t.

for all objects w and morphisms $f : x \to w$ and $g : y \to w$, there exists a unique morphism $h : x + y \to w$[❀] with $f = h \circ \mathrm{inj}_0$ and $g = h \circ \mathrm{inj}_1$.

In practice,



(iii) An object x is initial if for every object w, there exists the unique morphism $h : x \to w$.

(iv) An object x is final if for every object w, there exists the unique morphism $h : w \to x$.[✌]

(3) Note that all of these notions are defined purely in terms of morphisms, without referring to elements of objects. This is a bit like specifying a property of an abstract data type in terms of its operations, not in terms of its implementations.

[♣] often written as $\pi_0 : x \times y \to x$
[♦] often written as $\pi_1 : x \times y \to y$
[♥] same as $f \in \mathrm{Hom}\,[w, x]$
[♠] we often write h as $\langle f, g \rangle$. Reynolds writes it as $f \otimes g$.
[★] ! means uniqueness and ⇢ means existence
[❀] often written as $[f, g]$. Reynolds writes it as $f \oplus g$
[✌] h is often written as $!_w$ or !

(4) In the category Predom, the product of two predomains P_0 and P_1 that we studied in Chapter 4 is indeed a categorical product in (i). Also, the sum of P_0 and P_1 in Chapter 4 is indeed a categorical sum or co-product. The predomain of the singleton set $\left(\{\star\}, \sqsubseteq\right)$ is a terminal object. The predomain of the empty set $\left(\emptyset, \sqsubseteq\right)$ is an initial object.

(5) Consider the category corresponding to a pratially ordered set $(X, \sqsubseteq)$. Then, an object $x \in X$ is initial if and only if it is the least element in X. It is final if and only if it is the greatest element. For objects x, y in X, $x + y$ is the least upper bound of x and y, and $x \times y$ is the greatest lower bound of x and y.

Exercise 6.1. Prove (4) and (5).

6.4 Functor and Natural Transformation

(1) Intuitively, a functor is a structure-preserving map from one category to another. A natural transformation is a uniform map from one functor to another. In PL terms, a functor is a type constructor. (Recall that in this analogy, an object in a category is a type.) And a natural transformation is a polymorphic function.

(2) Let $\mathcal{C}$ and $\mathcal{D}$ be categories.

Definition. A functor F from $\mathcal{C}$ to $\mathcal{D}$ is a pair of two maps F_{obj} and F_{mor} s.t.

(i) F_{obj} maps objects in $\mathcal{C}$ to objects in $\mathcal{D}$.

(ii) for objects $x, y \in \text{Obj}\,(\mathcal{C})$[♣],

$$(F_{\text{mor}})_{x,y} \in \left[\, \text{Hom}_{\mathcal{C}}\,[x, y] \to \text{Hom}_{\mathcal{D}}\left[\, F_{\text{obj}}\,(x), F_{\text{obj}}\,(y)\,\right]\,\right]$$

(iii) F_{mor} preserves $\circ$ and id:

(a) for all objects x of $\mathcal{C}$,

$$(F_{\text{mor}})_{x,x}\,(\text{id}_x) = \text{id}_{F_{\text{obj}}(x)}$$

(b) for all morphisms $f \in \text{Hom}_{\mathcal{C}}\,[x, y]$ and $g \in \text{Hom}_{\mathcal{C}}\,[y, z]$,

$$(F_{\text{mor}})_{x,z}\,(g \circ f) = (F_{\text{mor}})_{y,z}\,(g) \circ (F_{\text{mor}})_{x,y}\,(f)$$

We use F to mean F_{obj} and $(F_{\text{mor}})_{x,y}$.

(3) To see common examples, we need to understand to product of two categories. When $\mathcal{C}$ and $\mathcal{D}$ are categories, the product category $\mathcal{C} \times \mathcal{D}$ is defined as follows:

- $\text{Obj}\,(\mathcal{C} \times \mathcal{D}) = \left\{(x, y) \;\middle|\; x \in \text{Obj}\,(\mathcal{C}) \wedge y \in \text{Obj}\,(\mathcal{D})\right\}$
- $\text{Hom}_{\mathcal{C}\times\mathcal{D}}\left[\,(u, v), (x, y)\,\right] = \left\{(f, g) \;\middle|\; f \in \text{Hom}_{\mathcal{C}}\,[u, x] \wedge g \in \text{Hom}_{\mathcal{D}}\,[v, y]\right\}$
- $(f, g) \circ \left(f', g'\right) = \left(f \circ f', g \circ g'\right)$

[♣] objects of $\mathcal{C}$

- $\mathrm{id}_{(x,y)} = (\mathrm{id}_x, \mathrm{id}_y)$

(4) The $\times, +, \perp$ operators on predomains are in fact functors.

- $(-)_\perp$: Predom $\to$ Predom
  $(-)_\perp (P) = P_\perp$
  $$(-)_\perp (f) = \lambda x. \begin{cases} f(x) & \text{if } f(x) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$
- $\times^{\clubsuit}$: Predom $\times^{\blacklozenge}$ Predom $\to$ Predom.
  $\times (P_0, P_1) = P_0 \times^{\blacktriangledown} P_1$
  $\times (f, g) = \lambda (x, y) . \big(f(x), g(y)\big) = f \times g$ $^{\spadesuit}$
- $+^{\bigstar}$: Predom $\times$ Predom $\to$ Predom.
  $+ (P_0, P_1) = P_0 +^{\circledcirc} P_1$
  $$+ (f, g) = \lambda (x, y) . \begin{cases} \mathrm{inj}_0 \big(f(u)\big) & \text{if } x = \mathrm{inj}_0 (u) \\ \mathrm{inj}_1 \big(g(v)\big) & \text{if } x = \mathrm{inj}_1 (v) \end{cases} = f + g$$ $^{\text{✌}}$

(5) One other important functor is the identity functor:

Id: Predom$\to$ Predom

$\mathrm{Id}(X) = X$

$\mathrm{Id}(f) = f$

(6) Let F, G be functors from $\mathcal{C}$ to $\mathcal{D}$.

Definition. A natural transformation η from F to G, denoted $\eta : F \overset{\bullet}{\to} G : \mathcal{C} \to \mathcal{D}$ or $\mathcal{C} \overset{F}{\underset{G}{\Downarrow \eta}} \mathcal{D}$, is a collection of morphisms in $\mathcal{D}$ indexed by objects in $\mathcal{C}$:

$$\big(\eta_x : F(x) \to G(x)\big)_{x \in \mathrm{Obj}(\mathcal{C})}$$

such that

(i) for each object x in $\mathcal{C}$, η_x is a $\mathcal{D}$-morphism from $F(x)$ to $G(x)$.

(ii) for every morphism $f : x \to y$ in $\mathcal{C}$ (i.e., $f \in \mathrm{Hom}_{\mathcal{C}}[x, y]$),

$$\begin{array}{ccc} F(x) & \xrightarrow{\eta_x} & G(x) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(y) & \xrightarrow[\eta_y]{} & G(y) \end{array}$$

This is called naturality condition.

$^{\clubsuit}$product functor
$^{\blacklozenge}$product of categories
$^{\blacktriangledown}$product of predomains
$^{\spadesuit}$Reynolds' notation (and standard notation) that we covered in Chapter 4
$^{\bigstar}$sum functor
$^{\circledcirc}$sum of predomains
$^{\text{✌}}$notation that we used in Chapter 4

(7) One intuition is that F and G are type constructors and η is a polymorphic function from F to G. Whenever we are given a type x, we have an instantiation of η at x, that is a function from the type $F(x)$ to the type $G(x)$.

The condition (i) says that η should type check. The condition (ii) says that what η does at x should be identical in a sense to what it does at y. In other words, η should bot depend much on its type parameter x. This is called uniformity.

(8) Here are a few examples.

(i) unit: $\mathrm{Id} \xrightarrow{\bullet} (-)_\perp : \mathrm{Predom} \to \mathrm{Predom}$

$$\mathrm{unit}_P \in [\, P \to P_\perp \,]$$
$$\mathrm{unit}_P\,(x) = x$$

(ii) Let fst be a functor from $\mathcal{C} \times \mathcal{D}$ to $\mathcal{C}$, defined by

$$\mathrm{fst}\,(x, y) = x \text{ (for objects)}$$
$$\mathrm{fst}\,(f, g) = f \text{ (for morphisms)}$$

Then,
$\pi_0 : (-) \times (-) \xrightarrow{\bullet} \mathrm{fst} : \mathrm{Predom} \times \mathrm{Predom} \to \mathrm{Predom}$

$$(\pi_0)_{P,P'} \in \left[\, P \times P' \to P \,\right]$$
$$(\pi_0)_{P,P'}\,(a, b) = a$$

Exercise 6.2.

Show that unit and π_0 are indeed natural transformations.

Exercise 6.3.

π_1, inj_0, and inj_1 are also natural transformations if we pick appropriate functors. Find such functors.

(9) Notice that all of unit, π_0, π_1, inj_0, and inj_1 do something very straightforward intuitively. They don't do any clever tricks. Naturality condition says in a sense that a natural transformation doesn't do anything clever. In more positive terms, it says that a natural transformation only performs canonical operations.

Chapter 7

Recursively Defined Domains

Chapter 10 of Tennent's book

7.1 Motivation

(1) One reason that we studied category theory is to understand a general principle behind the construction of recursively defined domains, such as the following Ω that you encountered before:

$$\Omega \simeq \left(\hat{\Sigma} + \mathbb{Z} \times \Omega + (\mathbb{Z} \to \Omega)\right)_{\perp}$$
$$\hat{\Sigma} \stackrel{\text{def}}{=} \Sigma \cup \{\text{abort}\} \times \Sigma \simeq \Sigma + \Sigma$$

(2) If we write the RHS of the above isomorphism as $F(\Omega)$, the formula says:

$$\Omega \simeq F(\Omega)$$

That is, Ω is a fixed point of F. In fact, Ω is not just a fixed point, but the best fixed point where "the best" means something very similar to "the least" in the standard fixed point theorem of the domain theory.

(3) We will generalize the standard least fixed point theorem of the domain theory and obtain a general categorical least fixed point theorem. This generalization closely follows the intuition that categories are generalized partially ordered sets (and functors are generalized monotone functions). Then, we will instantiate our generalization with a particular category constructed out of domains and a particular kind of continuous functions called embeddings.

7.2 ω-chain and co-limit of ω-chain

(1) Let's start by remembering ingredients that we needed when expressing the standard least fixed point theorem of the domain theory.

(i) A partially ordered set D has the least element.

(ii) Every chain in D has the last upper bound

(iii) A function f on D is continuous (i.e., monotone and chain-limit-preserving).

Then, the theorem says that f has the least fixed point x_0. That is, $f(x_0) = x_0$ and for all y s.t. $f(y) = y$, $x_0 \leq y$.[♣]

(2) In the categorical generalization of the theorem,

(i) D becomes a category $\mathcal{C}$;

(ii) $f \in [D \to D]$ becomes a functor $F : \mathcal{C} \to \mathcal{C}$;

(iii) x_0 becomes an object in $\mathcal{C}$;

(iv) the least element of D becomes the initial object of $\mathcal{C}$;

Note that the monotonicity of f[♦] translates to F's morphism map being type-checked with respect to its object map. (i.e., for all $g : x \to y$, $F(g) : F(x) \to F(y)$) This translated property is a part of the conditions for F being a functor. Thus, the monotonicity already holds for F in a sense.

(3) Ok, what remain? We still need to generalize

(i) chains

(ii) least upper bounds (or limiets) of chains

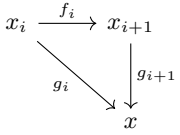
(iii) limit-preservation

(4) A ω-chain in a category $\mathcal{C}$ is a countably infinite sequence of objects $(x_0, x_1, x_2, \ldots)$ of $\mathcal{C}$ and a collection of morphisms $\{f_i : x_i \to x_{i+1}\}_{i \geq 0}$ in $\mathcal{C}$. The best way to understand this is to imagine the following figure:

$$x_0 \xrightarrow{f_0} x_1 \xrightarrow{f_1} x_2 \xrightarrow{f_2} \cdots$$

We use $\triangle$ to denote a ω-chain.

(5) A co-cone of an ω-chain $\triangle = \{(x_i, f_i)\}_{i \geq 0}$ is a pair of object x and a collection of morphisms $\{g_i : x_i \to x\}_{i \geq 0}$ such that for all $i \geq 0$, $g_{i+1} \circ f_i = g_i$, i.e., in picture,



commutes.

(6) Intuitively, an ω-chain is a generalized chain, and the object x of a co-cone $\left(x, \{g_i\}_i\right)$ of $\triangle$ is a generalized upper bound of the chain. Each $g_i : x_i \to x$ provides a way to view that x is larger than or equal to x_i. Meanwhile, each $f_i : x_i \to x_{i+1}$ provides a way to view that x_{i+1} is larger than or equal to x_i. The commutativity requirement says that these two views should be compatible.

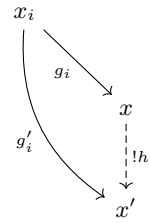
[♣] property that is a bit stronger than x_0 being the least fixed point

[♦] preservation of the $\sqsubseteq$ relation

(7) A co-cone of an ω-chain $\triangle = \left\{(x_i, f_i)\right\}_{i\geq 0}$ is co-limiting if for every co-cone $\left(x', \left\{g'_i\right\}_i\right)$ of $\triangle$, there exists a unique morphism $h : x \to x'$ such that for all i,

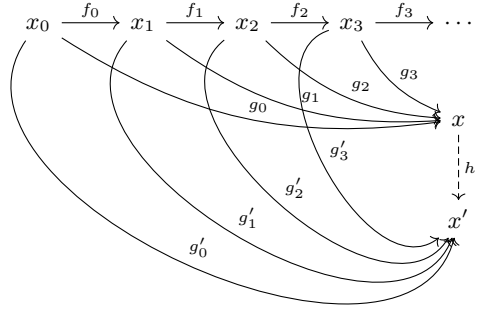
$$g'_i = h \circ g_i,$$

in a diagram,



Intuitively, the very existence of h says that x is smaller than or equal to x'. The commutativity and the uniqueness say that h's explanation about why x is smaller than or equal to x' follows automatically and canonically from the g_i and the g'_i.

(8) A co-limiting co-cone $\left(x, \{g_i\}_i\right)$ of an $\triangle$ is a generalization of the least upper bound of a chain. I usually imagine the following visual image whenever I work with an ω-chain, a co-cone, or a co-limiting co-cone:



Exercise 7.1.

Construct co-limiting co-cones of ω-chains in the category of Set. Do the same thing in the poset (partially ordered set) category $\left(2^{\mathbb{N}}, \subseteq\right)$ and in the category of predomains.

7.3 ω-continuous functor

(1) Let $\mathcal{C}$ and $\mathcal{D}$ be categories that have co-limiting co-cones[♣] for all ω-chains. We will call such categories as chain-complete categories.

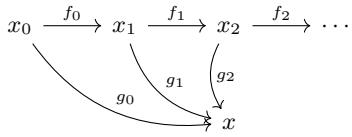
[♣] in other words, co-limits

(2) A functor $F : \mathcal{C} \to \mathcal{D}$ is ω-continuous if it maps a co-limit of an ω-chain to a co-limit of an ω-chain, that is,

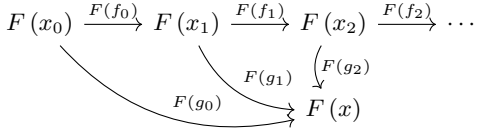
$$\begin{aligned}
&\text{for every } \omega\text{-chain } \triangle = \left\{(x_i, f_i)\right\}_{i\geq 0} \text{ in } \mathcal{C},\\
&\text{for every co-limit } \left(x, \{g_i\}_i\right) \text{ of } \triangle,\\
&\qquad\qquad \left(F(x), \left\{F(g_i)\right\}_i\right) \text{is a co-limit of}\\
&\qquad\qquad F(\triangle) = \left(\left\{F(x_i)\right\}_i, \left\{F(f_i)\right\}_i\right) \text{ in } \mathcal{D}.
\end{aligned}$$

(3) Intuitively, this ω-continuity of F means that F preserves the least upper bound of an increasing chain.

(4) Note that a functor F always maps a co-cone of an ω-chain to a co-cone of an ω-chain: visually,



category $\mathcal{C}$



category $\mathcal{D}$

If the diagram in $\mathcal{C}$ commutes, the diagram in $\mathcal{D}$ also commutes. This is because the preservation of $\circ$ and id by a functor implies that the functor maps every commuting diagram to a commuting diagram.

The situation is similar to the fact that a monotone function f from a predomain to a predomain maps an upper bound of a chain to an upper bound of a chain.

(5) However, the functoriality of F doesn't ensure that if $\left(x, \{g_i\}_i\right)$ is a co-limit of $\triangle$, so is $\left(F(x), \left\{F\left(g_i'\right)\right\}_i\right)$.

When F satisfies this additional property, we say that F is ω-continuous.

Exercise 7.2.

Show that the functor from Set to Set

$$\begin{aligned}
F(S) &= \mathbb{Z} \times S\\
F(f) &= \mathrm{id}_{\mathbb{Z}} \times f = \lambda(n, s) \,.\, \left(n, f(s)\right)
\end{aligned}$$

is ω-continuous.

7.4 Fixed point theorem

Theorem 7.1. Let $\mathcal{C}$ be a category with an initial object x_0. Assume that $\mathcal{C}$ is chain-complete (i.e., every ω-chain $\triangle$ in $\mathcal{C}$ has a co-limit). Then, for every ω-continuous functor $F : \mathcal{C} \to \mathcal{C}$, there exists an object x_{fix} in $\mathcal{C}$ and a morphism $\eta : F(x_{\text{fix}}) \to x_{\text{fix}}$ such that

(i) η is an isomorphism, i.e., $\exists$ a morphism $\psi : x_{\text{fix}} \to F(x_{\text{fix}})$ s.t. $\eta \circ \psi = \text{id}_{x_{\text{fix}}}$ and $\psi \circ \eta = \text{id}_{F(x_{\text{fix}})}$;

(ii) for every morphism $\eta' : F(y) \to y$, there exists a unique morphism $\rho : x_{\text{fix}} \to y$ such that

$$\begin{array}{ccc} F(x_{\text{fix}}) & \xrightarrow{\eta} & x_{\text{fix}} \\ {\scriptstyle F(\rho)}\downarrow & & \vdots\,{\scriptstyle \rho} \\ F(y) & \xrightarrow{\eta'} & y \end{array}$$

(1) Intuitively, the theorem says that $subsctextxfix$ is the least fixed point of F. The condition (i) says that x_{fix} is a fixed point. The condition (ii) says that it is the least fixed point.

(2) The proof is complex, but note that difficult. Very similar to the proof of the standard fixed point theorem of the domain theory. We will study only some parts of the proof.

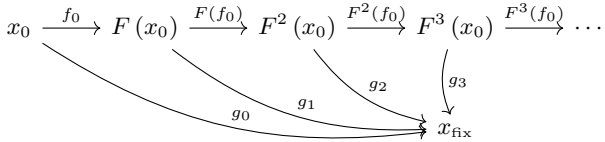
(3) The key part of the proof is to construct x_{fix}. Here we use the initial object x_0 of $\mathcal{C}$, the functoriality of F, and the chain completeness of $\mathcal{C}$ (They correspond to $\perp$, the monotonicity of a continuous function f and the chain completeness of a predomain D in the proof of the fixed-point theorem in domain theory).

We construct a chain:

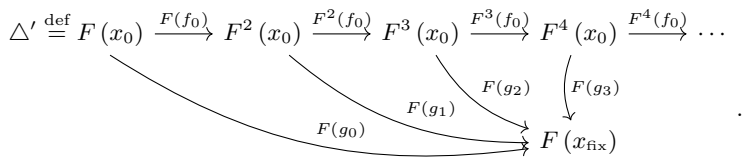
$$\triangle \stackrel{\text{def}}{=} x_0 \xrightarrow{f_0} F(x_0) \xrightarrow{f_1} F^2(x_0) \xrightarrow{f_2} F^3(x_0) \xrightarrow{f_3} \cdots$$

where f_0 is a unique morphism from the initial object x_0 to $F(x_0)$, and $f_i = F^i(f_0)$ is a unique morphism from $F^i(x_0)$ to $F^{i+1}(x_0)$.

Since $\mathcal{C}$ is chain-complete, there exists a co-limit $\left(x, \{g_i\}\right)$ of the chain $\triangle$ that we just built.

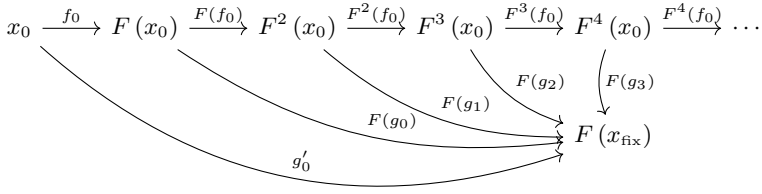


(4) Next we build $\eta : F(x_{\text{fix}}) \to x_{\text{fix}}$. Here we use the ω-continuity of F. Apply F to the diagram. This gives us



Since F is ω-continuous, this is a co-limiting co-cone of the chain $\triangle'$, which is the x_0-truncated version of $\triangle$.

(5) Because x_0 is the initial object, we can add x_0 to the diagram and get a co-limiting co-cone,



where g'_0 is a unique morphism from x_0 to $F(x_{\text{fix}})$. The leftmost triangle commutes because of the initiality of x_0.

Now we have two co-limits, x_{fix} and $F(x_{\text{fix}})$, of the same chain $\triangle$. One general result (which is easy to show) is that two co-limits of a chain are isomorphic, which means in our case that there exist morphisms $\eta : F(x_{\text{fix}}) \to x_{\text{fix}}$ and $\psi : x_{\text{fix}} \to F(x_{\text{fix}})$ such that

$$\psi \circ \eta = \text{id}_{F(x_{\text{fix}})} \quad \text{and} \quad \eta \circ \psi = \text{id}_{x_{\text{fix}}}.$$

We just proved (i) of the theorem. We leave the proof of (ii) as an exercise.

(6) Why is (ii) in the theorem useful? Because it allows us to define a morphism from x_{fix} to some y.

7.5 Famous example of the fixed point theorem

(1) One big motivation for developing domain theory was to find a solution of the following equation for space D:

$$D \simeq D \to D. \tag{7.1}$$

Such a space is needed (as you will see later in the course) to define a mathematical or denotational semantics of the untyped lambda calculus, which forms the core of most functional programming languages.

(2) But note that if D is a set and $D \to D$ is the set of all functions on D, the only solution of Eq. (7.1) is the singleton set (of course, in this case, $\simeq$ means some bijection between two sets). This is because if D contains more than one element, the cardinality of $[\, D \to D \,]$ is always strictly larger than that of D.

(3) Using domains, we can find a solution of Eq. (7.1) (using the fixed point theorem). But we have to be careful about defining a category on which we apply the theorem.

(4) Here is the category $\mathrm{Dom}^{\mathrm{EP}}$ that we use.

(i) Objects of $\mathrm{Dom}^{\mathrm{EP}}$ are domains; i.e., partially ordered set where all chains have least upper bounds and the least element exists.

(ii) Morphisms from a domain D to a domain D' are strict (i.e. $\bot$-preserving) continuous functions f from D to D' such that there exists a continuous function $g : D' \to D$ with

$$g \circ f = \mathrm{id}_D \quad \text{and} \quad f \circ g \sqsubseteq \mathrm{id}_{D'}.$$

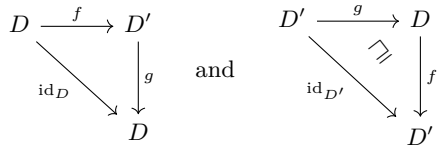
(iii) $\circ$ is the usual function composition.

(iv) id_D is the identity function on D.

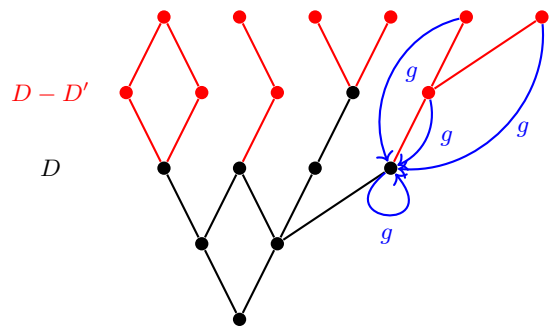
(5) Compared with the category Dom of domains and continuous functions, this category $\mathrm{Dom}^{\mathrm{EP}}$ has a rather unusual notion of morphisms. In $\mathrm{Dom}^{\mathrm{EP}}$, a morphism $f : D \to D'$ should be not just continuous, but also strict. More importantly, it should have $g : D' \to D$ such that

$$g \circ f = \mathrm{id}_D \quad \text{and} \quad f \circ g \sqsubseteq \mathrm{id}_{D'}.$$

In picture,



Intuitively, the existence of such g means that D' is built by putting additional elements above existing elements of D, and g maps all additional elements to their best under-approximations in D. Here is some picture that shows this intuition.



Where $D - D'$ is the set of additional elements in D' and g maps elements of D' to their best under-approximations in D. What this means is that a morphism $f : D \to D'$ is really saying that D' is larger than D.

(6) A morphism $f : D \to D'$ in $\mathrm{Dom}^{\mathrm{EP}}$ is called embedding and a corresponding $g : D' \to D$ is called projection.

If you happen to take a course on program analysis, this pair of embedding f and projection g is closely related to the Galois connection there.

Note That the definition doesn't say that there exists only one projection g for a given embedding f. That is, there may be multiple projections. But this doesn't happen.

Lemma 7.1. For every embedding $f : D \to D'$, and projections $g_1, g_2 : D' \to D$ for f, we have that

$$g_1 = g_2.$$

Proof. We will show that $g_1 \sqsubseteq g_2$. A similar argument can show the opposite inequality.

$$g_1 = \mathrm{id}_D \circ g_1 = (g_2 \circ f) \circ g_1 = g_2 \circ (f \circ g_1) \sqsubseteq g_2 \circ \mathrm{id}_{D'} = g_2.$$

□

We write f^{P} to denote the unique projection for an embedding f.

(7) The category $\mathrm{Dom}^{\mathrm{EP}}$ has an initial object, which is a singleton domain $\left(\{\bot\}, \sqsubseteq\right)$[♣]. It also has a co-limit for any ω-chain. We will not prove this. But we point out one important property of these co-limits.

Lemma 7.2. Consider the following co-cone of an ω-chain $\triangle$:

$$\triangle = \quad \begin{array}{ccccccc} D_0 & \xrightarrow{f_0} & D_1 & \xrightarrow{f_1} & D_2 & \xrightarrow{f_2} & \cdots \\ & \searrow^{h_0} & & \searrow^{h_1} & & \searrow^{h_2} & \\ & & & & D' & & \end{array}$$

Let h_i^{P} be the projection for the embedding h_i. Then, D' is co-limiting if and only if

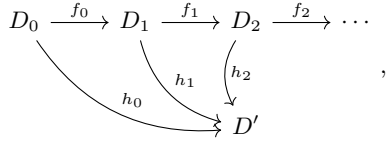
$$\bigsqcup_{i=0}^{\infty} \left(h_i \circ h_i^{\mathrm{P}}\right) = \mathrm{id}_{D'}.$$

Note that $h_i \circ h_i^{\mathrm{P}} \sqsubseteq \mathrm{id}_{D'}$. We can easily show that $\left\{h_i \circ h_i^{\mathrm{P}}\right\}_{i \geq 0}$ is an increasing chain in $\left[D' \underset{c}{\to} D'\right]$. The condition says that the least upper bound of the chain is $\mathrm{id}_{D'}$.

[♣] $x \sqsubseteq y$ iff $x = y$

This lemma says that the order on morphisms plays an important role in deciding whether D' is a co-limit or not. One important consequence is the following lemma.

Lemma 7.3. A functor $F : \mathrm{Dom}^{\mathrm{EP}} \to \mathrm{Dom}^{\mathrm{EP}}$ is ω-continuous if for every co-cone of an ω-chain $\triangle$



,

$$\bigsqcup_{i=0}^{\infty}\left(h_i \circ h_i^{\mathrm{P}}\right) = \mathrm{id}_{D'} \implies \bigsqcup_{i=0}^{\infty}\left(F\left(h_i\right) \circ F\left(h_i^{\mathrm{P}}\right)\right) = \mathrm{id}_{F\left(D'\right)}.$$

Proof. This is a direct consequence of the previous lemma. □

(8) Lemma 7.3 is our tool to check the ω-continuity of a functor F on $\mathrm{Dom}^{\mathrm{EP}}$. If this check passes, by the fixed point theorem, we know that there exists a domain D such that

$$F\left(D\right) \simeq D$$

(i) Our functor $F\left(\Omega\right) = \left(\Sigma + \Sigma + \mathbb{Z} \times \Omega + \left[\mathbb{Z} \to \Omega\right]\right)_{\perp}$ is an example of such a functor.

(ii) Another famous example is the following G that defines the function space.

$G\left(D\right) = \left[D \underset{c}{\to} D\right]$ ... the domain of continuous functions in D.

For every $f : D \to D'$,

$$G\left(f\right) : G\left(D\right) \to G\left(D'\right) \text{ i.e., } \left[D \underset{c}{\to} D\right] \to \left[D' \underset{c}{\to} D'\right]$$

$$G\left(f\right)\left(h\right) = f \circ h \circ f^{\mathrm{P}\clubsuit}.$$

Exercise 7.3. Prove that F and G are indeed ω-continuous.

Exercise 7.4. Prove Lemma 7.2. (This is not easy)

If you are familiar with program analysis and abstract interpretation, you might have noticed that there we do something similar when defining the abstract domain for function space. One thing to keep in mind is that in domain theory, $a \sqsubseteq b$ means that b is more informative than a, while in program analysis, $a \sqsubseteq b$ means that a is more informative than b.

$\clubsuit$We are using the projection of f here. If f were just a continuous function, we couldn't do it.

Chapter 8

The Lambda Calculus

8.1 Motivation

(1) Most programming languages support a mechanism for declaring functions and applying them to arguments. In fact, in functional languages, such as Ocaml, Haskell, Clojure and Scala (to some extent), function declaration and application (not state access and update) is the main device of computation.

(2) The lambda calculus is a simple formal language that lets us study principles behind function declaration and application without being distracted by the complexities of usual real-world programming languages. If forms the basis of many functional languages. Also, it can be used to define a notion of computability.

(3) One interesting construct of the lambda calculus is so-called lambda abstraction.

$$\lambda x.e$$

which denotes a function with formal argument x and body e. Nowadays most mainstream languages (c++, Java, Python, etc.) support this construct. The lambda abstraction is particularly useful when we use higher-order functions. For instance, to express

$$\int_0^1 dx \int_0^x dy\,(x+y)^2$$

in a programming language with the "integrate" primitive, we can write

$$\text{integrate}\left(0, 1, \lambda x.\ \text{integrate}\left(0, x, \lambda y.\ (x+y)\times(x+y)\right)\right)$$

using lambda abstraction. But without it, we should write

$$\text{let } f(x) = \left(\text{let } g(x) = (x+y)\times(x+y) \text{ in integrate}\,(0, x, g)\right) \text{ in integrate}\,(0, 1, f)$$

According to some rumor that I heard, one motivation for introducing lambda abstraction to java is to help people develop and use libraries with higher-order functions, in particular, collection libraries.

8.2 Syntax

$$\langle exp \rangle ::= \langle var \rangle \mid \langle exp \rangle \ \langle exp \rangle \ {}^{\clubsuit} \mid \lambda \ \langle var \rangle \ . \ \langle exp \rangle \ {}^{\blacklozenge}$$

(1) Examples

$$(\lambda x.x\ x)\,(\lambda z.z\ z)$$
$$\big(\lambda x.\,(\lambda y.y\ x)\ z\big)\ (z\ w)$$
$$(\lambda x.x\ x)\,(\lambda z.z)$$

- $\lambda x.\lambda y.x$ ... encoding of true in the lambda calculus
- $\lambda x.\lambda y.y$ ... encoding of false
- $\lambda f.\lambda x.x$ ... encoding of 0
- $\lambda f.\lambda x.f\ x$ ... encoding of 1
- $\lambda f.\lambda x.f\ \ (f\ x)$ ... encoding of 2
- $\lambda f.\lambda x.f\ \ \big(f\ \ (f\ x)\big)$ ... encoding of 3

(2) The set of free variables (in an expression of the lambda calculus), the substitution operator, and the α-equivalence (or renaming equivalence) for the lambda-calculus expressions are defined as you expect. Once you get the idea that the lambda operator in $\lambda x.e$ binds the variable x in the expression e, just as the quantifier $\forall$ in $\forall x.p$ binds the variable x in an assertion p.

(i)

$$\mathrm{FV}\,(v) = \{v\}$$
$$\mathrm{FV}\,(e_1\ e_2) = \mathrm{FV}\,(e_1) \cup \mathrm{FV}\,(e_2)$$
$$\mathrm{FV}\,(\lambda v.e) = \mathrm{FV}\,(e) \setminus \{v\}$$

(ii) δ is a substitution, i.e., a map from $\langle var \rangle$ to $\langle exp \rangle$:

$$v/\delta = \delta\,(v)$$
$$(e_1\ e_2)\,/\delta = \big(e_1/\delta\big)\ \big(e_2/\delta\big)$$
$$(\lambda v.e)\,/\delta = \lambda v_{\mathrm{new}}.\ \big(e/\,[\delta \mid v : v_{\mathrm{new}}]\big)$$
$$where v_{\mathrm{new}} \notin \bigcup_{w \in \mathrm{FV}(e) \setminus \{v\}} \mathrm{FV}\,\big(\delta\,(w)\big)$$

(iii) The renaming or change of bound variable means the operation of replacing an occurrence of $\lambda v.e$ by $\lambda v_{\mathrm{new}}.\ \big(e/_{v \to v_{\mathrm{new}}}\big)$, [♥] Two expressions e_1 and e_2 are <u>α-equivalent</u> or <u>renaming-equivalent</u> if we can obtain e_2 from e_1 by applying this renaming operation to some subexpressions of e_1 zero or multiple times. We write $e_1 \equiv e_2$ to denote their α-equivalence.

[♣] called application

[♦] called abstraction or lambda expression

[♥] This means the substitution that maps all variables to themselves except v, which it maps to v_{new}.

Example.

$$(\lambda x.x)\,(\lambda z.z) \equiv (\lambda x.x)\,(\lambda y.y)$$
$$(\lambda x.\lambda y.x) \equiv (\lambda y.\lambda x.y)$$

8.3 Reduction

(1) When we studied operational semantics, we used transition relation $\rightarrow$ to model one-step computation. We do something similar for the lambda calculus. We define a binary relation $\rightarrow \in \langle exp \rangle \times \langle exp \rangle$, called contraction relation, which models or formalizes single-step computation. Then, we will call the reflexive transitive closure of $\rightarrow$, i.e., $\rightarrow^*$, the reduction relation.

(2) Here is the definition of the contraction relation using inference rules.

(i) β-reduction

$$\frac{}{(\lambda v.e)\,e' \rightarrow e/_{v \rightarrow e'}}$$

(ii) Renaming

$$\frac{e_0 \rightarrow e_1 \qquad e_1 \equiv e_1'}{e_0 \rightarrow e_1'}$$

(iii) Contextual closure

$$\frac{e_0 \rightarrow e_1}{e_0' \rightarrow e_1'}$$

e_1' is obtained from e_0' by replacing one occurrence of e_0 in e_0' by e_1.

(3) The real change happens by the first rule, β-reduction. The second rule means that the contraction relation is defined on α-equivalence classes. The third rule says that any subexpression of a given e can be contracted by the β-reduction rule. Here are a few examples that may help you to see what goes on here.

$$(\lambda x.y)\,(\lambda z.z) \rightarrow y$$
$$(\lambda x.x\ x)\,(\lambda z.z) \rightarrow (\lambda z.z)\,(\lambda z.z) \rightarrow (\lambda z.z)$$
$$\big(\lambda x.\,(lambday.y\ x)\,z\big)\,(z\ w) \rightarrow (\lambda x.z\ x)\,(z\ w) \rightarrow z\ \ (z\ w)$$
$$\big(\lambda x.\,(lambday.y\ x)\,z\big)\,(z\ w) \rightarrow \big(\lambda y.y\,(z\ w)\big)\ \ z \rightarrow z\ \ (z\ w)$$

(4) Because of the third rule, the contraction relation is not deterministic. That is, $e_0 \rightarrow e_1$ and $e_0 \rightarrow e_2$ do not imply that $e_1 \equiv e_2$. For a counterexample, look at the example above. However, this nondeterminism comes from any nondeterministic constructs of the lambda calculus (which do not exist). The following theorem shows one consequence of this, and expresses that the contraction relation is essentially deterministic.

Property 8.1 (Church-Rosser Theorem)**.**
If $e \to^* e_0$[♣] and $e \to^* e_1$, then there exists e_2 s.t. $e_0 \to^* e_2$ and $e_1 \to^* e_2$.

(5) An expression e is a β-normal form if it cannot be contracted. Intuitively, such an expression denotes the final result of some computation, and the reduction relation $\to^*$ performs computation by transforming a given expression to normal form. Property 8.1 implies that every expression can be reduced to at most one normal-form expression modulo α-equivalence.

Property 8.2.
If $e_0 \to^* e_1$, $e_0 \to e_2$ and e_1 and e_2 are β-normal forms, then $e_1 \equiv e_2$. (i.e. e_1 and e_2 are α-equivalent)

Proof. By Property 8.1, $\exists e_3$ s.t. $e_1 \to^* e_3$ and $e_2 \to^* e_3$. Since e_1 and e_2 are β-normal forms, $e_1 \equiv e_3$ and $e_2 \equiv e_3$. $\therefore e_1 \equiv e_2$. □

(6) One natural question is whether we can find a good strategy to use the nondeterminism in the third "Contextual closure" rule, so that if an expression e can be reduced to a normal form, this strategy indeed transforms e to such a normal-form expression.

To see this issue, consider the following two reduction sequences:

$$\begin{aligned}(\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right) &\to \lambda v.v\\ (\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right) &\to (\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right)\\ &\to \cdots\end{aligned}$$

Only the first gives an expression in the normal form.

(7) The normal-order reduction is a particular way of using the "Contextual closue" rule. It picks a redex[♦] (β-redex) in an expression e that is not included in any other redex. Also, if there are multiple such outermost redexes, it picks the leftmost one. In our example, the normal-order reduction doesn't pick the redex $(\lambda x.x\ x)\,(\lambda x.x\ x)$ because it is included in the redex $(\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right)$. The normal-order reduction is also called outermost leftmost reduction.

Property 8.3. If $e \to^* e'$ for some normal-form e', then $e \to^*_{\text{normal}} e'$ where $\to^*_{\text{normal}}$ means the contraction relation of the normal-order reduction.

8.4 Normal-order evaluation and eager evaluation

(1) In functional languages, we use restricted versions of the reduction relation to specify how function calls should be handled. We will look at two well-known restrictions used in Haskell and Ocaml, and call them normal-order evaluation and eager evaluation. Note that we use the world "evaluation" instead of "reduction" or "contraction".

[♣] reflexive transitive closure of $\to$
[♦] reducible expression

(2) Both normal-order evaluation and eager evaluations are defined for closed expressions only, i.e., expressions that do not have any free variables. Also, they are formalized as big-step semantics where the evaluation relation $\Rightarrow$ transforms an expression to a final result in one go, instead of in multiple steps in the reduction relation. Finally, these evaluations do not contract any subexpressions inside lambda. Thus, their results might not be normal forms. They will instead be a canonical form.

(3) Normal-order evaluation $\Rightarrow$:

$$e^{\clubsuit} \Rightarrow z^{\blacklozenge}$$

A canonical form z is a lambda expression.

Canonical Forms

$$\frac{}{\lambda v.e \Rightarrow \lambda v.e}$$

Application (β-evaluation)

$$\frac{e \Rightarrow \lambda v.e'' \qquad \left(e''/_{v \to e'}\right) \Rightarrow z}{e\ e' \Rightarrow z}$$

Property 8.4. For any closed expression e and canonical form z, $e \to^* z$ iff $e \Rightarrow z$.

Exercise 8.1.

Try to prove Property 8.4. The proof is in the page 203 / 204 of the textbook. Reading this proof helped me understand the $\Rightarrow$ relation better.

(4) Intuitively, the normal-order evaluation works by postponing the evaluation of the arguments of a function. The arguments get evaluated when they are needed. However, this evaluation strategy may be inefficient because it may repeat the evaluation of one argument. For instance, look at the following example.

$$(\lambda x.x\ x)\left((\lambda y.y)\,(\lambda z.z)\right)$$

In the normal-order evaluation, the redex $(\lambda y.y)\,(\lambda z.z)$ gets contracted twice; because of the two occurrences of x in the body of $\lambda x.x\ x$.

(5) The eager evaluation takes a different approach. It evaluates the arguments of a function before applying the function. Most programming languages implement this eager evaluation strategy.

(6) Eager evaluation (formalized by) $\Rightarrow_{\mathrm{E}}$:

$$e^{\heartsuit} \Rightarrow_{\mathrm{E}} z^{\spadesuit}$$

$^{\clubsuit}$closed expression
$^{\blacklozenge}$expression in the canonical form (i.e., lambda expression)
$^{\heartsuit}$closed expression
$^{\spadesuit}$canonical form

Canonical Forms

$$\frac{}{\lambda v.e \Rightarrow_{\mathrm{E}} \lambda v.e}$$

Application (β_{E}-evaluation)

$$\frac{e \Rightarrow_{\mathrm{E}} \lambda v.e'' \qquad e' \Rightarrow_{\mathrm{E}} z' \qquad \left(e''/_{v \to z'}\right) \Rightarrow_{\mathrm{E}} z}{e\ e' \Rightarrow z}$$

Exercise 8.2.

(a) Show that

$$(\lambda x.x\ x)\left((\lambda y.y)\,(\lambda z.z)\right) \Rightarrow_{\mathrm{E}} (\lambda z.z)\,.$$

How many times does the redex $(\lambda y.y)\,(\lambda z.z)$ get contracted?

(b) What can we get on the RHS of $\Rightarrow_{\mathrm{E}}$ below?

$$(\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right) \Rightarrow_{\mathrm{E}} ???$$

What about $\Rightarrow$?

$$(\lambda u.\lambda v.v)\left((\lambda x.x\ x)\,(\lambda x.x\ x)\right) \Rightarrow ???$$

(c) Which one do you like more between $\Rightarrow$ and $\Rightarrow_{\mathrm{E}}$? Why?

8.5 Denotational semantics

(1) Interpreting the lambda calculus denotationally is not easy. It was one of the long-standing open problems in the 1960's, until Scott solved it using the techniques from the domain theory (which Scott himself developed partly for this purpose).

(2) To understand why it was an open problem, let's try to interpret expressions in the lambda calculus (denotationally) using sets and functions. This trial will fail as we explain shortly and will show the challenge of handling the fact that in the lambda calculus, functions can cat on themselves.

The first thing that we should do (in this trial) is to find an appropriate space S (which is this case is a set) for the meanings of the expressions. Let's suppose that we somehow managed to find such S. Then, S should include all functions on S that may be denoted by expressions in the lambda calculus. Let's make a slightly stronger but nicer assumption that

$$[\,S \to S\,]^{\clubsuit} \subseteq S$$

we will now show that S is the singleton set. This is because the assumption implies that every function f on S has a fixed point, which can happen only if S is a singleton set. What fixed point does a function $f \in [\,S \to S\,]$ have?

$^{\clubsuit}$set of all functions on S

Here is an answer for the question. Let p a function on S s.t. $p(x) = f\left(x(x)\right)$ for all $x \in [S \to S] \subseteq S$. Such p exists. Then $p(p) = f\left(p(p)\right)$. So, $p(p)$ is a fixed point of f.

We have just shown that $[S \to S] \not\in S$ for any set S that contains more than one element.

(3) What should we do? We need to use the domain theory and the categorical tools, in particular general categorical fixed-point theorem and the category $\mathrm{DOM}^{\mathrm{EP}}$, which consists of domains and a particular kind of strict conditions functions called embeddings. If you are curious about these, look at the notes on "5. Famous Example of the Fixed point Theorem" (6 Nov 2018). Using these tools, we can find domains D_1, D_2, D_3 and V_2, V_3 s.t.

(i) $D_1{}^{\clubsuit} \simeq^{\blacklozenge} \left[D_1 \underset{c}{\to}{}^{\blacktriangledown} D_1 \right]$

(ii) $D_2 \simeq \left[D_2 \underset{c}{\to} D_2 \right] \times \left[D_2 \underset{c}{\to} D_2 \right]$

$D_3 \stackrel{\text{def}}{=} (V_3)_\perp$

(iii) $V_2 \stackrel{\text{def}}{=} \left[D_2 \underset{c}{\to} D_2 \right]$

$V_3 \simeq \left[V_3 \underset{c}{\to} (V_3)_\perp \right]$

(4) Note that D_1, D_2 and V_3 are solutions of slightly different recursive domain equations. Why do we consider three such equations, instead of one? It is because the contraction relation, the normal-order evaluation and the eager evaluation provide three different meanings to the expressions in the lambda calculus, and these equations capture these differences. D_1 is for the contraction relation, D_2 and V_2 are for the normal-order evaluation, and D_3 and V_3 are for the eager evaluation.

(5) As before, understanding these domains is the key to understand the denotational semantics of the lambda calculus under three different notions of computations:

(i) the contraction relation

(ii) the normal-order evaluation

(iii) the eager evaluation

The equations for $(D_2, V_2)\, and\, (D_3, V_3)$ indicate that under (ii) and (iii), we need to differentiate expressions that don't terminate and those that do and become lambda expressions. D_2 and D_3 are domains for all expressions, and V_2 and V_3 are domains

---

$^{\clubsuit}$also denoted by D_∞ in other textbooks

$^{\blacklozenge}$isomorphism between domains

$^{\blacktriangledown}$continuous functions. We often omit c but here we wrote it explicitly to emphasize the fact that we are considering continuous functions only here.

for the latter kind of expressions. Sometimes V_2 and V_3 are called domains for values, and D_2 and D_3 are called domains for computations. Note that we don't make this kind of distinction for D_1. For instance, in D_2 and D_3, $\bot$ is different from the constant function that always return $\bot$. (i.e. $\lambda x.\bot$) On the other hand, in D_1, they are regarded the same. (In D_1, $\lambda x.\bot$ is the least element up to the isomorphism $D_1 \simeq \left[D_1 \underset{c}{\to} D_1 \right]$.)

Why is D_1's way of defining $\bot$ different from D_2 and D_3's? Because the normal-order evaluation and the eager evaluation are do not reduce subexpressions under the lambda abstraction ($\lambda x.e$), whereas the contraction relation do reduce such subexpressions.

(6) Now let's try to understand the difference between (D_2, V_2) and (D_3, V_3). Rewriting isomorphisms and definitions slightly can help us to see this difference more easily.

$$D_2 \overset{\text{def}}{=} (V_2)_\bot \qquad V_2 \simeq \left(\left[D_2 \underset{c}{\to} D_2 \right] \right)_\bot$$

$$D_3 \overset{\text{def}}{=} (V_3)_\bot \qquad V_3 \simeq \left[V_3 \underset{c}{\to} D_3 \right]$$

The key difference lies in the fact that V_2 has a function space $\left[\underline{D_2} \underset{c}{\to} D_2 \right]$, whereas V_3 has a function space $\left[\underline{V_3} \underset{c}{\to} D_3 \right]$. The argument domain for the normal-order evaluation is that for computations, while the argument domain for the eager evaluation is that for values. This difference comes from the fact that in the eager evaluation, we pass only canonical forms (which are lambda expressions) to functions as arguments, while in the normal-order evaluation, we pass any expressions as function arguments. So, when we use the normal-order evaluation, variables may be bound to expressions that may denote any computations. But if we use the eager evaluation, variables should be bound to lambda expressions or more generally canonical forms that denote values.

(7) Here is the denotational semantics for the contraction relation.

$$D_1 \underset{\psi}{\overset{\phi}{\rightleftarrows}} \left[D_1 \underset{c}{\to} D_1 \right]$$

$$[\![-]\!] \in \left[\langle exp \rangle \to \left[D_1^{\langle var \rangle^\clubsuit} \underset{c}{\to} D_1 \right] \right] \qquad \eta \in D_1^{\langle var \rangle} \ldots \text{called environment}$$

$$[\![v]\!]\,\eta = \eta\,(v) \qquad [\![e_1\ e_2]\!]\,\eta = \phi\left([\![e_1]\!]\,\eta\right)\left([\![e_2]\!]\,\eta\right)$$

$$[\![\lambda x.e]\!]\,\eta = \psi\left(\lambda a \in D_1.\,[\![e]\!]\,[\eta \mid x : a]\right)$$

$^\clubsuit D_1^{\langle var \rangle} = \left[\langle var \rangle \to D_1 \right]$ ordered pointwise

Property 8.5 (Textbook 10.8)**.**
Well-defined. That is, $\lambda a \in D_1.\, [\![e]\!]\,[\eta \mid x]\,a$ is continuous.

Property 8.6 (Textbook 10.12 and 10.13)**.**

$$\text{If } e_0 \to e_1, \text{ then } [\![e_0]\!]\,\eta = [\![e_1]\!]\,\eta \text{ for all } \eta \in D_1^{\langle var \rangle}.$$

The contraction relation preserves the semantics. Note that this implies that the reduction relation $\to^*$ also preserves the semantics.

(8) Here are the denotational semantics for the normal-order evaluation and that for the eager evaluation.

normal-order evaluation:

$$D_2 = (V_2)_\perp \qquad V_2 \underset{\psi}{\overset{\phi}{\rightleftarrows}} \left[D_2 \underset{c}{\to} D_2 \right] \times \left[V_2 \underset{c}{\to} V_2 \right]$$

$$[\![-]\!]_{\mathrm{n}} \in \left[\langle exp \rangle \to \left[D_2^{\langle var \rangle} \underset{c}{\to} D_2 \right] \right]$$

$$[\![v]\!]_{\mathrm{n}}\,\eta = \eta\,(v)$$

$$[\![e_0\ e_1]\!]_{\mathrm{n}}\,\eta = \begin{cases} \perp & \text{if} \quad [\![e_0]\!]_{\mathrm{n}}\,\eta = \perp \\ \phi\left([\![e_0]\!]_{\mathrm{n}}\,\eta\right)\left([\![e_1]\!]_{\mathrm{n}}\,\eta\right) & \text{otherwise} \end{cases}$$

$$[\![\lambda x.e]\!]_{\mathrm{n}}\,\eta = \psi\left(\lambda a \in D_2.\, [\![e]\!]_{\mathrm{n}}\,[\eta \mid x : a]\right)$$

eager evaluation:

$$D_3 = (V_3)_\perp \qquad V_3 \underset{\psi}{\overset{\phi}{\rightleftarrows}} \left[V_3 \underset{c}{\to} D_3 \right]$$

$$[\![-]\!]_{\mathrm{e}} \in \left[\langle exp \rangle \to \left[V_3^{\langle var \rangle} \underset{c}{\to} D_3 \right] \right]$$

$$[\![v]\!]_{\mathrm{e}}\,\eta = \eta\,(v)$$

$$[\![e_0\ e_1]\!]_{\mathrm{e}}\,\eta = \begin{cases} \perp & \text{if} \quad [\![e_0]\!]_{\mathrm{e}}\,\eta = \perp \text{ or } [\![e_1]\!]_{\mathrm{e}}\,\eta = \perp \\ \phi\left([\![e_0]\!]_{\mathrm{e}}\,\eta\right)\left([\![e_1]\!]_{\mathrm{e}}\,\eta\right) & \text{otherwise} \end{cases}$$

$$[\![\lambda x.e]\!]_{\mathrm{e}}\,\eta = \psi\left(\lambda a \in V_3.\, [\![e]\!]_{\mathrm{e}}\,[\eta \mid x : a]\right)$$

Both semantics are well-defined. They validate the normal-order evaluation relation and the eager evaluation relation. That is,

$$\left(e \Rightarrow e'\right) \quad \Longrightarrow \quad [\![e]\!]_{\mathrm{n}}\,\eta = \left[\!\left[e'\right]\!\right]_{\mathrm{n}}\,\eta \text{ for all } \eta \in D_2^{\langle var \rangle}$$

$$\left(e \Rightarrow_{\mathrm{E}} e'\right) \quad \Longrightarrow \quad [\![e]\!]_{\mathrm{e}}\,\eta = \left[\!\left[e'\right]\!\right]_{\mathrm{e}}\,\eta \text{ for all } \eta \in V_3^{\langle var \rangle}$$

Chapter 9

An Eager Functional Language

9.1 Motivation

(1) In Chapter 8, we learned the lambda calculus and the eager evaluation for it. They form the basis of most (or nearly all) call-by-value functional programming languages such as OCaml, Clojure, Scala, Scheme etc. But developing such a real eager functional programming language involves much more than including the lambda calculus and adopting the eager evaluation. The goal of this chapter and the subsequent few chapters is to understand these additional things.

(2) In the chapter, we will mainly study two topics related to the following questions:

(i) In order to help solve most real-world computational problems naturally, a functional programming language should include constants and operations for primitive types, such as `int`, and mechanisms for building data structures. How can we do this? How should we change the abstract grammar, the notion of canonical form, evaluation relation and denotational semantics?

(ii) Also, we need to support recursive definitions. What should we do?

(3) We will focus on answering these questions. In Chapter 10 of the textbook, Reynolds explains a lot more than what we will cover. He even gives well-known list-manipulation examples of functional programming. If you are interested in them, have a look at chapter 10.

9.2 Constants and primitive operations, basic (dynamic) types

(1) Recall the syntax of the lambda calculus and the notion of canonical form.

$$\langle exp \rangle ::= \langle var \rangle \mid \langle exp \rangle \ \langle exp \rangle \mid \lambda \ \langle var \rangle \ . \ \langle exp \rangle$$

$$\langle cfm \rangle ::= \langle funcfm \rangle$$

$$\langle funcfm \rangle ::= \lambda \ \langle var \rangle \ . \ \langle exp \rangle$$

Here we expressed the notion of canonical forms using the abstract grammar.

We also recall the rules for evaluation; where we omit E in $\Rightarrow_{\mathrm{E}}$ since we are interested in eager evaluation only here.

Canonical Forms

$$\frac{}{\lambda v.e \Rightarrow_{\mathrm{E}} \lambda v.e}$$

β_{E}-evaluation

$$\frac{e \Rightarrow_{\mathrm{E}} \lambda v.e'' \qquad e' \Rightarrow_{\mathrm{E}} z' \qquad \left(e''/_{v \to z'}\right) \Rightarrow_{\mathrm{E}} z}{e\ e' \Rightarrow z}$$

(z and z' are canonical forms. In general, we use z with primes, subscripts or superscripts to denote canonical forms.)

(2) Adding constants, primitive operations, and those for basic type constructors (tuple, alternative / sum) amounts to extending $\langle exp \rangle$, $\langle cfm \rangle$, and $\Rightarrow$. Extending $\langle cfm \rangle$ with more cases intuitively means that we make the language support values other than lambda expressions. The extension of $\Rightarrow$ encodes the meanings of newly introduced constants and operations.

(3) We add four new kinds of values. That is, we change the grammar for $\langle cfm \rangle$ as follows:

$$\begin{aligned}
\langle cfm \rangle &::= \langle funcfm \rangle \\
&\mid \langle intcfm \rangle \mid \langle boolcfm \rangle \\
&\mid \langle tuplecfm \rangle \mid \langle altcfm \rangle
\end{aligned}$$

$$\begin{aligned}
\langle intcfm \rangle ::= &\ \ldots \mid \text{-2} \mid \text{-1} \mid 0 \mid 1 \\
&\mid 2 \mid \ldots
\end{aligned}$$

$$\langle boolcfm \rangle ::= \text{true} \mid \text{false}$$

$$\langle tuplecfm \rangle ::= \langle\ \langle cfm \rangle\ , \ldots , \langle cfm \rangle\ \rangle$$

$$\langle altcfm \rangle ::= @\ \langle tag \rangle\ \langle cfm \rangle$$

$$\langle tag \rangle ::= 0 \mid 1 \mid 2 \mid \ldots$$

This extension means that expressions in this language denote five different kinds of values. Another important point is that we only include operations for constructing tuples and alternatives, not those for destructing them; such as projection and case operation. In a sense, this comes from the fact that destructors represent unfinished computation, and we regard something as value or canonical from when it represents completed computation.

(4) Next we extend $\langle exp \rangle$ with appropriate constants and operations so that we can write expressions denoting computations with those newly introduced canonical forms, i.e., integers, booleans, tuples, and alternatives.

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle var \rangle \mid \langle exp \rangle\ \langle exp \rangle \mid \lambda\, \langle var \rangle\ .\ \langle exp \rangle \\
& \mid & 0 \mid 1 \mid 2 \mid \ldots \mid \text{-}\ \langle exp \rangle \mid \langle exp \rangle \overset{+}{\underset{\text{rem}}{\overset{-}{\times}\ \div}} \langle exp \rangle \\
& \mid & \text{true} \mid \text{false} \mid \neg\ \langle exp \rangle \mid \langle exp \rangle \overset{\wedge}{\vee}{\underset{\Leftrightarrow}{\Rightarrow}} \langle exp \rangle \\
& \mid & \text{if } \langle exp \rangle \text{ then } \langle exp \rangle \text{ else } \langle exp \rangle \mid \langle exp \rangle \overset{=}{\neq}{\overset{<}{\leq}}{\overset{>}{\geq}} \langle exp \rangle \\
& \mid & \langle\ \langle exp \rangle\ , \ldots, \langle exp \rangle\ \rangle\ ^{\clubsuit} \mid \langle exp \rangle\ .\ \langle tag \rangle\ ^{\blacklozenge} \\
& \mid & @\ \langle tag \rangle\ \langle exp \rangle\ ^{\blacktriangledown} \mid \text{sumcase } \langle exp \rangle \text{ of } (\ \langle exp \rangle, \ldots, \langle exp \rangle\ )\ ^{\spadesuit} \\
& \mid & \text{error} \mid \text{typeerror}
\end{array}
$$

We already said that introducing four new kinds of canonical forms amounts to introducing four new runtime (or dynamic) types to the language, namely, int, bool, tuple and alternative. For tuple and alternative, we have operations for constructing tuple and alternative expressions, and also operations for destructing them. This is a typical pattern that appears repeatedly when one designs a new programming language. When he or she adds a static or dynamic/runtime type to a language, he or she also introduces appropriate constructors and destructors for the type to the language.

(5) Evaluation relation $\Rightarrow$ should also be extended to incorporate intended semantics of newly added operations. For integers and booleans, we change $\Rightarrow$ according to the standard semantics of primitive operations. Hence, we give rules for only some operations. Other cases are similar.

$$\frac{e \Rightarrow i}{-e \Rightarrow \hat{-} i} \qquad \frac{e \Rightarrow b}{\neg e \Rightarrow \hat{\neg} b}$$

$$\frac{e \Rightarrow i \qquad e' \Rightarrow i'}{e \text{ op } e' \Rightarrow i \ \hat{\text{op}}\ i'} \quad \begin{array}{l}\text{where op}\in\{+,-,\times,=,\neq,<,\leq,>,\geq\} \\ \text{or } \big(\text{op}\in\{\div,\text{rem}\} \text{ and } i'\neq 0\big)\end{array}$$

$$\frac{e \Rightarrow b \qquad e' \Rightarrow b'}{e \text{ op } e' \Rightarrow b \ \hat{\text{op}}\ b'} \quad \text{when op}\in\{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$$

$$\frac{e \Rightarrow \text{true} \qquad e' \Rightarrow z}{\text{if } e \text{ then } e' \text{ else } e'' \Rightarrow z} \qquad \frac{e \Rightarrow \text{false} \qquad e'' \Rightarrow z}{\text{if } e \text{ then } e' \text{ else } e'' \Rightarrow z}$$

Note that i and b are integer and boolean constants, respectively. Also, I used $\hat{}$ to emphasize mathematical operations.

For operations for tuples and alternatives, we include rules for constructors that just evaluate their arguments, and those for destructors that convert constructed tuples

---

$^{\clubsuit}$constructing a tuple
$^{\blacklozenge}$projection destructing a tuple
$^{\blacktriangledown}$constructing an alternative
$^{\spadesuit}$case analysis for destructing an alternative

and alternatives back to some of their components.

$$\frac{e_0 \Rightarrow z_0 \quad \ldots \quad e_{n-1} \Rightarrow z_{n-1}}{\langle e_0, \ldots, e_{n-1}\rangle \Rightarrow \langle z_0, \ldots, z_{n-1}\rangle} \qquad \frac{e \Rightarrow \langle z_0, \ldots, z_{n-1}\rangle}{e.k \Rightarrow z_k}\ \text{when } k<n$$

$$\frac{e \Rightarrow z}{@\ k\ e \Rightarrow @\ k\ z} \qquad \frac{e \Rightarrow @\ k\ z \qquad e_k\ z \Rightarrow z'}{\text{sumcase } e \text{ of } (e_0, \ldots, e_{n-1}) \Rightarrow z'}\ \text{when } k<n$$

9.3 Recursion

(1) We include letrec:

$$\langle exp\rangle ::= \ \ldots \mid \text{letrec } \langle var\rangle \equiv \lambda\ \langle var\rangle\ .\ \langle exp\rangle \text{ in } \langle exp\rangle$$

(letrec $v \equiv \lambda w.e$ in e') defines a recursive function v and performs e' with v bound to this recursive function. Thus,

$$\text{FV}\left(\text{letrec } v \equiv \lambda w.e \text{ in } e'\right) = \left(\left(\text{FV}\left(e\right) \setminus \{w\}\right) \cup \text{FV}\left(e'\right)\right) \setminus \{v\}$$

This means that the occurrence of v in e denotes $\lambda w.e$, the recursive function being defined here, not the value of a free variable v.

(2) This construct imposes two important constraints. First, recursively defined entities should be functions like $\lambda w.e$. For instance, we can't do

$$\text{letrec } v \equiv \langle 1, v\rangle \ \text{ in } e$$

which defines an infinite tuple $v = \left\langle 1, \left\langle 1, \langle 1, \ldots\rangle\right\rangle\right\rangle$.

Second, the RHS of $\equiv$ should be a canonical form. For instance, the following is not allowed.

$$\text{letrec } v \equiv \left(\lambda x.\left(\lambda y.3\right)\right)\left(\lambda z.z\right) \ \text{ in } e$$

Both restrictions are included because we use the eager evaluation. In a programming language based on the normal-order evaluation such as Haskell, we don't need to impose those restrictions. When we discuss denotational semantics, you will understand where these restrictions come from.

(3) Since adding letrec doesn't add a new kind of denotable values by expressions, we don't change $\langle cfm\rangle$. (Although we don't show, letrec can be expressed using lambda expressions and applications.) But we need to add a rule for evaluating letrec expressions. Here is the rule.

$$\frac{e'/_{v\to\left(\lambda w.e/_{v\to\text{letrec } v\equiv\lambda w.e \text{ in } v}\right)}{}^{\clubsuit} \Rightarrow z}{\text{letrec } v \equiv \lambda w.e \text{ in } e' \Rightarrow z}{}_{\blacklozenge}$$

$^{\clubsuit}$unrolling of recursive call v in $\lambda w.e$

$^{\blacklozenge}$execution of e' with v bound to its definition

(4) Programming exercise:

Suppose that we represent binary trees with integer leaves using alternative and tuple as follows.

- @ 0 n ... for a terminal note (or leaf) labelled by the integer n.
- @ 1 $\langle l, r\rangle$ ... for a non-terminal node with left subtree l and right subtree r.

Write a program that sums all the integers in a given tree.

Answer:

$$\text{letrec add} \equiv \lambda t.\ \text{sumcase } t \text{ of } \left(\lambda n.n,\ \lambda t'.\ \Big(\text{add}\left(t'.0\right) + \text{add}\left(t'.1\right)\Big)\right) \text{ in add}$$

9.4 Denotational Semantics

(1) Recall the denotational semantics for the lambda calculus, and the eager evaluation, in particular domains used and the form of the semantics function $[\![-]\!]$.

$$\begin{aligned} D &\stackrel{\text{def}}{=} V_\perp \\ V &\simeq V \underset{c}{\to} D \quad , \quad \text{more precisely, } V \underset{\psi}{\overset{\phi}{\rightleftarrows}} V \to D \\ [\![-]\!] &\in \Big[\langle exp\rangle \to V^{\langle var\rangle} \to D\Big] \end{aligned}$$

Since wee added four new kinds of values, we should change $\simeq$ so that the isomorphism says V consists of not just continuous functions but also those new values. Also, we have to change $V_\perp$ to account for errors and failures of the runtime typechecker.

(2) We use the following V and V_* (which corresponds to D above) and change the form of $[\![-]\!]$ accordingly.

$$\begin{aligned} V_* &\stackrel{\text{def}}{=} \big(V + \{\text{error},\ \text{typeerror}\}\big)_\perp \\ V &\underset{\psi}{\overset{\phi}{\rightleftarrows}} V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}} + V_{\text{tuple}} + V_{\text{alt}} \\ V_{\text{int}} &\stackrel{\text{def}}{=} \mathbb{Z} \\ V_{\text{bool}} &\stackrel{\text{def}}{=} \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\} \\ V_{\text{fun}} &\stackrel{\text{def}}{=} V \underset{c}{\to} V_* \\ V_{\text{tuple}} &\stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} V^n = \bigcup_{n=0}^{\infty} V \times \cdots \times V \\ V_{\text{alt}} &\stackrel{\text{def}}{=} \mathbb{N} \times V \end{aligned}$$

(i) Don't be confused between V here and V in (1). They are different. Also, note the parallel between the definition of $\langle cfm \rangle$ and the isomorphism for V here. This isomorphism is in a sense a denotational way of saying there are five kinds of canonical forms (or values).

(ii) V_* extends V not just with $\perp$ but also with error and typeerror, so that the semantics can express such errors.

(iii) In general, the denotational semantics of an eager functional language has the form:

$$[\![-]\!] \in \left[\langle exp \rangle \to V^{\langle var \rangle} \to T^{\clubsuit}(V) \right]$$

and interprets functions using

$$V \underset{c}{\to} T(V)$$

This indicates that variables always get bound to values/canonical forms, not to arbitrary computations.

(3) The actual definition of $[\![-]\!]$ is involved, but in a sense straightforward. The only things to be noteworthy are the uses of f_* and $g_{\theta *}$ for $\theta \in \{\text{int}, \text{bool}, \text{fun}, \text{tuple}, \text{alt}\}$.

$$f \in \left[V \underset{c}{\to} V_* \right] \qquad f_* \in \left[V_* \underset{c}{\to} V_* \right]$$

$$f_*(a) = \begin{cases} f(b) & \text{if } a = \langle 0, b \rangle^{\blacklozenge} \\ a & \text{otherwise} \end{cases}$$

$$g \in \left[V_\theta \underset{c}{\to} V_* \right] \qquad g_{\theta *} \in \left[V_* \underset{c}{\to} V_* \right]$$

$$\left(\theta \in \{\text{int}, \text{bool}, \text{fun}, \text{tuple}, \text{alt}\}\right)$$

$$g_{\theta *}(a) = \begin{cases} g(b) & \text{if } a = \langle 0, b \rangle \text{ and } \exists i, b' \text{ s.t. } \substack{b = \psi\langle i, b' \rangle \\ \text{and } b' \in V_\theta} \\ \langle 1, \text{typeerror} \rangle & \text{if } a = \langle 0, b \rangle \text{ and } \neg \left(\substack{b = \psi\langle i, b' \rangle \\ \text{and } b' \in V_\theta} \right) \\ a & \text{otherwise} \end{cases}$$

Intuitively, when $g_{\theta *}$ is applied to an element in V (a value, not an error or $\perp$), i.e. $\langle 0, b \rangle$, it first does runtime typechecking and finds out whether b is a value of type θ. If not, $g_{\theta *}$ returns a typeerror $\langle 1, \text{typeerror} \rangle$. If yes, on the other hand, it casts b to a value b' of type θ, and runs g on b'. Thus, $g_{\theta *}$ does type-testing and type-casting.

$^{\clubsuit}T$: domain constructor like $(-)_\perp$ and $(-)_*$

$^{\blacklozenge}$the first component of $V + \{\text{error}, \text{typeerror}\}$

(4) Here is the definition of $[\![-]\!]$. We present only some cases. Look at the textbook for the complete definition.

$$[\![-]\!] \in \left[\langle exp \rangle \to V^{\langle var \rangle \clubsuit} \to V_* \right]$$

$$[\![v]\!]\,\eta = \left\langle 0, \eta\,(v) \right\rangle$$

$$[\![e_0\ e_1]\!]\,\eta = \begin{cases} [\![e_0]\!]\,\eta & \text{if } [\![e_0]\!]\,\eta \in \left\{ \bot, \langle 1, \text{error} \rangle, \langle 1, \text{typeerror} \rangle \right\} \\ \langle 1, \text{typeerror} \rangle & \text{else if } [\![e_0]\!]\,\eta = \langle 0, b \rangle \text{ but } \neg \begin{pmatrix} \exists b' \text{ s.t. } b = \psi\langle 2, b' \rangle \\ \wedge\ b' \in V_{\text{fun}} \end{pmatrix} \\ [\![e_1]\!]\,\eta & \text{else if } [\![e_1]\!]\,\eta \in \left\{ \bot, \langle 1, \text{error} \rangle, \langle 1, \text{typeerror} \rangle \right\} \\ b'\,(a) & \text{else if } \begin{pmatrix} [\![e_0]\!]\,\eta = \langle 0, b \rangle \text{ and } \begin{pmatrix} \exists b' \text{ s.t. } b = \psi\langle 2, b' \rangle \\ \wedge\ b' \in V_{\text{fun}} \end{pmatrix} \\ \text{and } [\![e_1]\!]\,\eta = \langle 0, a \rangle \end{pmatrix} \end{cases}$$

More succinctly, we can write:

$$[\![e_0\ e_1]\!]\,\eta = \left(\lambda f.\, \left(\lambda z.f\,(z) \right)_* \left([\![e_1]\!]\,\eta \right) \right)_{\text{fun}*} \left([\![e_0]\!]\,\eta \right)^{\blacklozenge}$$

$$[\![\lambda v.e]\!]\,\eta = \left\langle 0, \psi \left(\left\langle 2, \lambda z.\, [\![e]\!]\,[\eta \mid v]\, z \right\rangle \right) \right\rangle$$

$$[\![\langle e_0, \cdots, e_{n-1} \rangle]\!]\,\eta = \left(\lambda z_0. \cdots \left(\lambda z_{n-1}. \left\langle 0, \langle z_0, \cdots, z_{n-1} \rangle \right\rangle \right)_* \left([\![e_{n-1}]\!]\,\eta \right) \cdots \right)_* \left([\![e_0]\!]\,\eta \right)$$

$$[\![e.k]\!]\,\eta = \left(\lambda t.\ \text{if } k \in \text{dom}\,(t)^{\blacktriangledown} \begin{cases} \text{then } \langle 0, t_k \rangle \\ \text{else } \langle 1, \text{typeerror} \rangle \end{cases} \right)_{\text{tuple}*} \left([\![e]\!]\,\eta \right)$$

$$[\![@\ k\ e]\!]\,\eta = \left(\lambda z. \left\langle 0, \psi \left(\left\langle 4, \langle k, z \rangle \right\rangle \right) \right\rangle \right)_* \left([\![e]\!]\,\eta \right)$$

$$[\![\text{sumcase } e \text{ of } (e_0, \ldots, e_{n-1})]\!]\,\eta$$

$$= \left(\lambda\, \langle k, z \rangle \ \text{if } k < n \begin{cases} \text{then } \left(\lambda f.f\,(z) \right)_{\text{fun}*} \left([\![e_k]\!]\,\eta \right) \\ \text{else } \langle 1, \text{typeerror} \rangle \end{cases} \right)_{\text{alt}*} \left([\![e]\!]\,\eta \right)$$

$$[\![k]\!]\,\eta = \left\langle 0, \psi \left(\langle 0, k \rangle \right) \right\rangle$$

$$[\![e_0 + e_1]\!]\,\eta = \left(\lambda i. \left(\lambda i'. \left\langle 0, \psi \langle 0, i + i' \rangle \right\rangle \right)_{\text{int}*} \left([\![e_1]\!]\,\eta \right) \right)_{\text{int}*} \left([\![e_0]\!]\,\eta \right)$$

$$[\![\text{typeerror}]\!]\,\eta = \langle 1, \text{typeerror} \rangle$$

$$[\![\text{true}]\!]\,\eta = \left\langle 0, \psi \left(\langle 1, \mathbf{tt} \rangle \right) \right\rangle$$

$\clubsuit$ often written as E to emphasize that it is the set of environments

$\blacklozenge$ λf, λz and $f\,(z)$: function definitions and application in math, $(\ )_{\text{fun}*}$: typechecking and typecasting

$\blacktriangledown$ t consists of at least $k+1$ components

Exercise 9.1.

Suppose that $[\![e]\!]\,\eta = \bot$. What are $[\![e + \text{true}]\!]\,\eta$ and $[\![\text{true} + e]\!]\,\eta$?

(5) Although complex, the definition of $[\![e]\!]$ is so far is in a sense straightforward because it is almost automatically derived from our informal understanding of e's meaning, the form of $[\![-]\!]$ and the definition of V and V_* (i.e., semantic types), and the idea of inserting type-testing and type-casting using $(-)_{\theta *}$. But the case for recursive function

$$\left[\!\left[\text{letrec } v \equiv \lambda u.e \text{ in } e'\right]\!\right]\eta$$

is not that straightforward. To see why, consider the following naive (and incorrect) attempt:

$$[\![\text{letrec } v \equiv \lambda u.e \text{ in } e']\!]\,\eta = [\![\,e'\,]\!]\left[\eta \;\middle|\; v : Y_{V_*}\,(F)\right]$$

$$F : V_* \to V_* \qquad F\,(b) = [\![\lambda u.e]\!]\,[\eta \mid v : b]$$

What's wrong with this? $\left[\eta \;\middle|\; v : Y_{V_*}\,(F)\right]$ and $[\eta \mid v : b]$ are not environments because $Y_{V_*}\,(F)$ and b might not be values; i.e., elements of the form $\langle 0, c\rangle$ for some $c \in V$. This mathematical problem comes from the mismatch between the semantics here and the operation semantics (i.e., evaluation relation)

What should we do? We use the fact that $[\![\lambda u.e]\!]\,\eta$ is always of the form

$$\left\langle 0, \psi\left(\left\langle 2, \lambda z.\,[\![e]\!]\,[\eta \mid u : z]\right\rangle\right)\right\rangle \in V \underset{c}{\to} V_* = V_{\text{fun}}.$$

Also, note that although V is not a domain, $V_{\text{fun}} = V \underset{c}{\to} V_*$ is a domain. Thus, every continuous function on $V \underset{c}{\to} V_*$ has the least fixed point. This means that we can define

$$F \in \left[\left[V \underset{c}{\to} V_*\right] \to \left[V \underset{c}{\to} V_*\right]\right]$$
$$F\,(f)\,(z) = [\![e]\!]\,[\eta \mid v : f \mid u : z]$$

Then, we can interpret the recursive definition as follows:

$$\left[\!\left[\text{letrec } v \equiv \lambda u.e \text{ in } e'\right]\!\right]\eta = \left[\!\left[e'\right]\!\right]\left[\eta \;\middle|\; v : \left\langle 0, \psi\left(\left\langle 2, Y_{V \underset{c}{\to} V_*}\,(F)\right\rangle\right)\right\rangle\right]$$

If v is not bound to a function or v is not bound to a canonical form, we cannot use this interpretation. This is one of the reasons that we consider only this restricted form of recursive definitions in eager functional programming languages. OCaml adopted the same restriction.

Chapter 10

Continuations in a Functional Language

10.1 Motivation

(1) Intuitively, a continuation means the remaining computation. For instance, when evaluating the subexpression $3+4$ in $\Big(9 \times \big(8 + (3+4)\big)\Big)$, we have the continuation that denotes $9 \times \big(8 + [\,]\big)$, which expresses what we should do after calculating $3+4$.

(2) Continuations appear in multiple forms in programming languages. First, they are used in a particular style of programming, called continuation passing style. In this style of programming, called CPS, operators like $+$ and $\times$ take continuation parameter κ additionally. For instance,

$$+_{\mathrm{CPS}}(m, n, \kappa) = \kappa(m+n)$$
$$\times_{\mathrm{CPS}}(m, n, \kappa) = \kappa(m \times n).$$

Using these new operations, we can write $\Big(9 \times \big(8 + (3+4)\big)\Big)$ as follows:

$$+_{\mathrm{CPS}}\Big(3, 4, \lambda r_1.\, +_{\mathrm{CPS}}\big(8, r_1, \lambda r_2.\, \times_{\mathrm{CPS}}(9, r_2, \lambda r_3.r_3)\big)\Big).$$

Second, continuations are first-class values, and they are used to express highly generalized `goto`s in expressive higher-order programming languages, such as Scheme. Those languages include the construct, `callcc`, and often `throw` as well. The former is like label in C and C++, and the latter is like `goto`. When used wisely, these constructs lead to really cool programming examples that alter the flow of computation in an intricate way. They are often used to implement coroutine, backtracking, scheduler, generator, etc.

Third, continuations are also a powerful tool for building a compiler for functional languages. Some compilers transform programs or expressions to those in continuation passing style in the early phase of compilation. After this CPS-transformation, expressions no longer depend on whether we use eager evaluation or normal-order evaluation. Both evaluations give the same result when applied to CPS-transformed expressions.

Fourth, continuations form a powerful tool in the denotational semantics. In fact, they frequently feature in mathematics. Let V be the predomain for values that we looked at in the previous chapter. Then, semantically, continuations are elements in

$$\left[V \underset{c}{\to} A \right]$$

for some domain A. If you studied functional analysis or Banach space or Hilbert space before, you might have seen the dual of a vector space V over $\mathbb{R}$,

$$V^* = [\, V \to_{\text{linear}} \mathbb{R} \,],$$

which consists of linear maps from V to $\mathbb{R}$. V^* can be understood as a space of continuations on V.

(3) In this chapter, we will primarily study continuations as new language feature (second point) and as a tool in the semantics (fourth point). But we will say a few words on the CPS transformation (third point).

(4) Another big part of this chapter is a semantic version of defunctionalization, a technique to replace higher-order functions by records. This is one of the key techniques in compilation. Also, a large number of PL researchers, especially those working on object-oriented languages, use defunctionalized denotational semantics.

10.2 Continuation Semantics

(1) One way to understand the idea of continuation is to rewrite the semantics of the eager functional programming language in the previous chapter, using continuation. In this setting, continuations are continuous functions from V to V_*.

$$V_{\text{cont}} \stackrel{\text{def}}{=} \left[V \underset{c}{\to} V_* \right]$$

They represent the rest of computation. If we provide the value a of the current computation step to a continuation κ, the computation (represented by $\kappa\,(a)$) will perform all the remaining computation steps and output the final result, which is the value of $\kappa\,(a)$.

(2) Let's define this continuation semantics more formally. Recall the semantic domains and predomains that we used in the semantics of the eager functional language in

the previous chapter.

$$V_* = \big(V + \{\text{error}, \text{typeerror}\}\big)_\perp$$

$$V \rightleftarrows^{\phi}_{\psi} V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}} + V_{\text{tuple}} + V_{\text{alt}}$$

$$V_{\text{int}} = \mathbb{Z} \qquad V_{\text{bool}} = \mathbb{B} \qquad V_{\text{fun}} = V \underset{c}{\to} V_*$$

$$V_{\text{tuple}} = \bigcup_{n \geq 0} V^n \qquad V_{\text{alt}} = \mathbb{N} \times V$$

$$[\![-]\!] \in \left[\langle exp \rangle \underset{c}{\to} \left[V^{\langle var \rangle} \underset{c}{\to} V_* \right] \right]$$

(3) The key idea of continuation semantics is to add a new input to $[\![-]\!]$ that represents continuation, and also to add a new parameter to each function that again represents continuation. This means the following two changes:

$$V_{\text{cont}} = V \underset{c}{\to} V_*$$

$$V_{\text{fun}} = V \underset{c}{\to} \left[\underline{V_{\text{cont}}} \underset{c}{\to} V_* \right]$$

$$[\![-]\!] \in \left[\langle exp \rangle \underset{c}{\to} \left[V^{\langle var \rangle} \underset{c}{\to} \left[\underline{V_{\text{cont}}} \underset{c}{\to} V_* \right] \right] \right]$$

The remaining parts of the semantic predomains and domains are defined in the same way as before.

(4) Altering V_{fun} and the form of $[\![-]\!]$ has a huge impact on the defining clauses of $[\![-]\!]$. When defining $[\![-]\!]$, we now have to specify how a given continuation is used and modified. Observed this change in the following definition of $[\![-]\!]$.

$$[\![v]\!]\, \eta\, \kappa = \kappa\, \big(\eta\, (v)\big)$$

$$\Big[\!\Big[e\ e' \Big]\!\Big]\, \eta\, \kappa = [\![e]\!]\, \eta\ \Big(\lambda f.\ \Big[\!\Big[e' \Big]\!\Big]\, \eta\ (\lambda z. f\ z\ \kappa) \Big)_{\text{fun}}$$

$$\vdots$$

Here $(-)_\theta$ is similar to $(-)_{\theta *}$ that we looked at before, but it doesn't deal with $\perp$ and errors. That is, given $f \in V_\theta \to V_*$,

$$f_\theta \in V_\theta \to V_*$$

$$f_\theta\, (a) = \begin{cases} \langle 1, \text{typeerror} \rangle & \text{if } \neg \begin{pmatrix} \exists i, b \text{ s.t. } b \in V_\theta \\ \wedge\ a = \psi\big(\langle i, b \rangle\big) \end{pmatrix} \\ f\, (b) & \text{if } \begin{pmatrix} \exists i, b \text{ s.t. } b \in V_\theta \\ \wedge\ a = \psi\big(\langle i, b \rangle\big) \end{pmatrix} \end{cases}$$

$$\vdots$$

$$
\begin{aligned}
[\![\lambda v.e]\!]\,\eta\,\kappa &= \kappa\left(\psi\left\langle 2, \lambda a.\lambda\kappa'.\,[\![e]\!]\,[\eta \mid v : a]\ \kappa'\right\rangle\right)\\
[\![n]\!]\,\eta\,\kappa &= \kappa\left(\psi\left\langle 0, n\right\rangle\right)\\
[\![-e]\!]\,\eta\,\kappa &= [\![e]\!]\,\eta\ \left(\lambda i.\ \kappa\left(\psi\left\langle 0, -i\right\rangle\right)\right)\\
[\![e_1 + e_2]\!]\,\eta\,\kappa &= [\![e_0]\!]\,\eta\ \left(\lambda i.\,[\![e_1]\!]\,\eta\ \left(\lambda i'.\ \kappa\left(\psi\left\langle 0, i + i'\right\rangle\right)\right)_{\text{int}}\right)_{\text{int}}\\
\left[\!\left[\text{if } e \text{ then } e' \text{ else } e''\right]\!\right]\,\eta\,\kappa &= [\![e]\!]\,\eta\ \left(\lambda b.\ \text{if } b \text{ then } \left[\!\left[e'\right]\!\right]\,\eta\,\kappa \text{ else } \left[\!\left[e''\right]\!\right]\,\eta\,\kappa\right)_{\text{bool}}\\
[\![\text{true}]\!]\,\eta\,\kappa &= \kappa\left(\psi\left\langle 1, \mathbf{tt}\right\rangle\right)\\
[\![\langle e_0, e_1\rangle]\!]\,\eta\,\kappa &= [\![e_0]\!]\,\eta\ \left(\lambda a_0.\,[\![e_1]\!]\,\eta\ \left(\lambda a_1.\kappa\left(\psi\left\langle 3, \langle a_0, a_1\rangle\right\rangle\right)\right)\right)\\
[\![e.k]\!]\,\eta\,\kappa &= [\![e]\!]\,\eta\ \left(\lambda t.\ \text{if } k \in \text{dom}\,(t)\ \begin{cases}\text{then } \kappa\,(t_k)\\ \text{else } \langle 1, \text{typeerror}\rangle\end{cases}\right)_{\text{tuple}}\\
\left[\!\left[\text{letrec } v \equiv \lambda u.e \text{ in } e'\right]\!\right]\,\eta\,\kappa &= \left[\!\left[e'\right]\!\right]\,[\eta \mid v : Y\ F]\ \kappa\\
F &\in \left[V_{\text{fun}} \underset{c}{\to} V_{\text{fun}}\right]\\
F\,(f_0)\,(a)\left(\kappa'\right) &= [\![e]\!]\,[\eta \mid u : a \mid v : f_0]\ \kappa'
\end{aligned}
$$

We omit a few definitions. You can find them in the page 254-255 of the textbook.

(5) Note that whenever we interpret an expression that includes more than one immediate subexpression, such as $e_0 + e_1$, we construct a new continuation for the subexpressions that will not be evaluated next, such as

$$\left(\lambda i.\,[\![e_1]\!]\,\eta\ \left(\lambda i'.\ \kappa\left(\psi\left\langle 0, i + i'\right\rangle\right)\right)_{\text{int}}\right)_{\text{int}}.$$

Intuitively, this means that the semantics is very explicit about evaluation order.

(6) This semantics can be expressed as syntactic transformation call CPS transformation. Let $[\![-]\!]_d$ be the direct semantics that we studied in the previous chapter. Consider an expression e and a fresh variable v_{cont}. Then, this transformation has the following property:

$$[\![e]\!]\,\eta\,\kappa \text{ “}=\text{” }^{\clubsuit}\ [\![\underline{\text{CPS}}\,(e, v_{\text{cont}})]\!]_d\,[\eta \mid v_{\text{cont}} : \kappa]$$

As mentioned before, this CPS transformation is often used by a compiler as a preprocessing step.

$^{\clubsuit}$equal when no errors

(7)
$$\underline{\mathrm{CPS}}\,(v, v_{\mathrm{cont}}) = v_{\mathrm{cont}}\,(v)$$
$$\underline{\mathrm{CPS}}\,\left(e\ e', v_{\mathrm{cont}}\right) = \underline{\mathrm{CPS}}\,\left(e, \lambda f.\ \underline{\mathrm{CPS}}\,\left(e', \lambda u.\ f\ u\ v_{\mathrm{cont}}\right)\right)$$
$$\underline{\mathrm{CPS}}\,(\lambda v.e, v_{\mathrm{cont}}) = v_{\mathrm{cont}}\,\left(\lambda v.\ \lambda v'_{\mathrm{cont}}.\ \underline{\mathrm{CPS}}\,\left(e, v'_{\mathrm{cont}}\right)\right)$$
$$\underline{\mathrm{CPS}}\,(n, v_{\mathrm{cont}}) = v_{\mathrm{cont}}\,(n)$$
$$\underline{\mathrm{CPS}}\,(-e, v_{\mathrm{cont}}) = \underline{\mathrm{CPS}}\,\left(e, \lambda u.\ v_{\mathrm{cont}}\,(-u)\right)$$
$$\underline{\mathrm{CPS}}\,(e_0 + e_1, v_{\mathrm{cont}}) = \underline{\mathrm{CPS}}\,\left(e_0, \lambda u_0.\ \underline{\mathrm{CPS}}\,\left(e_1, \lambda u_1.\ v_{\mathrm{cont}}\,(u_0 + u_1)\right)\right)$$
$$\underline{\mathrm{CPS}}\,\left(\text{if } e \text{ then } e' \text{ else } e'', v_{\mathrm{cont}}\right) = \underline{\mathrm{CPS}}\left(e, \lambda b.\ \text{if } b \begin{cases} \text{then } \underline{\mathrm{CPS}}\,\left(e', v_{\mathrm{cont}}\right) \\ \text{else } \underline{\mathrm{CPS}}\,\left(e'', v_{\mathrm{cont}}\right) \end{cases}\right)$$
$$\underline{\mathrm{CPS}}\,(\mathrm{true}, v_{\mathrm{cont}}) = v_{\mathrm{cont}}\,(\mathrm{true})$$
$$\underline{\mathrm{CPS}}\,\left(\langle e_0, e_1\rangle, v_{\mathrm{cont}}\right) = \underline{\mathrm{CPS}}\,\left(e_0, \lambda u_0.\ \underline{\mathrm{CPS}}\,\left(e_1, \lambda u_1.\ v_{\mathrm{cont}}\,\left(\langle u_0, u_1\rangle\right)\right)\right)$$
$$\underline{\mathrm{CPS}}\,(e.k, v_{\mathrm{cont}}) = \underline{\mathrm{CPS}}\,\left(e, \lambda u.\ v_{\mathrm{cont}}\,(u.k)\right)$$
$$\underline{\mathrm{CPS}}\,\left(\text{letrec } v \equiv \lambda u.e \text{ in } e', v_{\mathrm{cont}}\right)\ ^{\clubsuit}$$
$$= \text{letrec } v \equiv \lambda u.\ \lambda v'_{\mathrm{cont}}.\ \underline{\mathrm{CPS}}\,\left(e, v'_{\mathrm{cont}}\right) \text{ in } \underline{\mathrm{CPS}}\,\left(e', v_{\mathrm{cont}}\right) \text{ for fresh } v_{\mathrm{cont}}.$$

(8) Note that all function calls after the CPS transformation are the applications of continuation variables to parameters. Since such variables represent the rest of computation, no calls leave anything to be done after they are complete. Thus, such calls can be implemented as `jump`, not as procedure call, by a compiler. Also, as mentioned before, the CPS-transformed programs produce the same result regardless of whether we are using eager evaluation or normal-order evaluation. These observations indicate that CPS-transformed programs or expressions are simpler than the original ones.

(9) The transformation in (7) can be obtained systematically from the continuation semantics by removing η and all the embeddings and converting κ to the variable v_{cont}. This is because they are closely related.

10.3 Callcc and throw

(1) Some programming languages allow continuations to be denotable values, and provide language constructs for manipulating continuation values.

(2) Semantically, this means that we change V as follows:

$$V \overset{\phi}{\underset{\psi}{\rightleftarrows}} V_{\mathrm{int}} + V_{\mathrm{bool}} + V_{\mathrm{fun}} + V_{\mathrm{tuple}} + V_{\mathrm{alt}} + \underline{V_{\mathrm{cont}}}$$

Syntactically, it often involves adding the following two constructs:

$$\langle exp\rangle ::= \text{callcc } \langle exp\rangle \mid \text{throw } \langle exp\rangle\ \langle exp\rangle$$

$^{\clubsuit}$Sorry, I'm less certain about this case. (editor: seems fine?)

callcc expects a function as its argument. $\big(\text{callcc } (\lambda f.e)\big)$ reifies the current continuation, binds f to it, and executes e. The bound continuation f can be invoked by throw, as in (throw f 3). This calls the continuation f with the value 3.

(3) Here are the semantic clauses for callcc and throw:

$$[\![\text{callcc } e]\!]\,\eta\,\kappa = [\![e]\!]\,\eta\left(\lambda f.\ f\left(\psi\left\langle 5, \kappa^{\clubsuit}\right\rangle\right)\kappa\right)_{\text{fun}}$$

$$\left[\!\left[\text{throw } e\ e'\right]\!\right]\,\eta\,\kappa^{\spadesuit} = [\![e]\!]\,\eta\left(\lambda\kappa'.\ \left[\!\left[e'\right]\!\right]\,\eta\,\kappa'^{\heartsuit}\right)_{\text{cont}}$$

Intuitively, in (callcc $\lambda\kappa.\cdots$throw $\kappa\ 3\cdots$), callcc $\lambda\kappa$ can be viewed as putting a label denoted by κ, and throw κ 3 can be understood as a `goto` to this label.

(4) What are the results of the following expressions?

(i) $\text{callcc } \big(\lambda\kappa.\ 2 + \text{throw } \kappa\ (3 \times 4)\big)$

(ii) $\Big(\text{callcc } \big(\lambda\kappa.\ \lambda x.\ \text{throw } \kappa\ (\lambda y.\ x + y)\big)\Big)\ 6$

The first example can be understood as skipping some part of computation. The second shows how we can repeat the computation of some part of an expression using continuation.

(5) The next example is likely very hard to understand because it uses features not explained so far, and it is also quite tricky. The example is from the page 290 of the textbook. Imagine that we would like to implement a routine backtrack that takes a function and tries the function with a parameter amb$^{\spadesuit}$ representing a nondeterministic choice between true and false. It collects the results of all those choices and returns a list of all those results. For instance,

$$\text{backtrack}\left(\lambda\text{amb.}\ \begin{array}{l}\text{if amb}\,\langle\rangle^{\star}\ \text{then}\ \big(\text{if amb}\,\langle\rangle\ \text{then 0 else 1}\big)\\ \text{else}\ \big(\text{if amb}\,\langle\rangle\ \text{then 2 else 3}\big)\end{array}\right)$$

should return

$$@\,1\left\langle 3,\ @\,1\left\langle 2,\ @\,1\left\langle 1,\ @\,1\left\langle 0,\ @\,0\,\langle\rangle\right\rangle\right\rangle\right\rangle,$$

which is often written as

$$3 :: 2 :: 1 :: 0 :: \text{nil}$$

representing the list of 3, 2, 1, and 0. Note that these are all the possible outcomes of the parameter function to backtrack. To implement backtrack with callcc and throw, we need a few more features in our language.

$^{\clubsuit}$current continuation copied
$^{\spadesuit}$ignored
$^{\heartsuit}$continuation obtained from e is used instead
$^{\spadesuit}$editor: ambiguous
*empty tuple

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \text{mkref } \langle exp \rangle \ ^{\clubsuit} \\
& | & \text{val } \langle exp \rangle \ ^{\blacklozenge} \\
& | & \langle exp \rangle := \langle exp \rangle \ ^{\blacktriangledown} \\
& | & \langle exp \rangle =_{\text{ref}} \langle exp \rangle \ ^{\spadesuit}
\end{array}
$$

Syntactic sugar.

$$
\begin{aligned}
\text{nil} &\stackrel{\text{def}}{=} @\, 0\, \langle\rangle \\
e :: e' &\stackrel{\text{def}}{=} @\, 1\, \langle e, e' \rangle \\
\text{listcase } e \text{ of } (e_1, e_2) &\stackrel{\text{def}}{=} \text{sumcase } e \text{ of } \Big(\lambda v.e_1, \lambda v.\, \big((e_2 v.0)\, v.1\big)\Big) \\
\text{let } v \equiv e \in e' &\stackrel{\text{def}}{=} \big(\lambda v.e'\big)\, e \\
e;\ e' &\stackrel{\text{def}}{=} \text{let } v \equiv e \text{ in } e' \qquad \text{for fresh } v
\end{aligned}
$$

$$
\begin{aligned}
\text{backtrack} \stackrel{\text{def}}{=} \lambda f.\ & \text{let } rl \equiv \text{mkref}\,(\text{nil}) \ \text{ in} \\
& \text{let } cl \equiv \text{mkref}\,(\text{nil}) \ \text{ in} \\
& rl := f\,\Big(\lambda u.\text{callcc } \big(\lambda k.\ (cl := k :: \text{val } cl)\,;\ \text{true}\big)\Big) :: \text{val } rl\ ; \\
& \text{listcase } (\text{val } cl\,) \text{ of } \big(\text{val } rl, \lambda c.\ \lambda r.\ (cl := r\ ;\ \text{throw } c\ \text{false})\big)
\end{aligned}
$$

Editor's note on the backtrack function

The backtrack function is a bit tricky to understand, so the editor will try to explain it.

(i) rl is a reference to a list of results, and cl is a reference to a list of continuations.

(ii) f is applied to a function that uses callcc (call with current continuation) to capture the current continuation k. This continuation k is added to the list of continuations cl along with the value true. The continuation represents a choice in the computation. If the function f decides to backtrack, it can invoke one of these continuations to return to the state represented by that continuation.

(iii) After f has been applied, the listcase operation examines the list of results rl. If cl is empty, the listcase operation returns the list of results rl. If there are any continuations in cl, one is removed and invoked (throw c false) and its associated computation is resumed. This effectively backtracks to the point where callcc captured that continuation, and the computation tries a different path by returning false instead of true.

$^{\clubsuit}$allocates a memory cell, initialized it with $\langle exp \rangle$ and returns the reference to the cell.
$^{\blacklozenge}$dereferences a reference
$^{\blacktriangledown}$updates a reference
$^{\spadesuit}$reference equality check

10.4 Deriving a First-order Semantics

(Semantic version of defunctionalization)

(1) The continuation semantics and the direct semantics both use functions so heavily, sometimes even higher-order functions, i.e., functions that take functions as parameters. Can we define a semantics that avoids the use of such functions, or at least minimizes the use of higher-order functions?

More concretely, recall the definitions of predomains and domains involved in the continuation semantics:

$$V_* = \big(V + \{\text{error}, \text{typeerror}\}\big)_\perp$$

$$V \overset{\phi}{\underset{\psi}{\rightleftarrows}} V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}} + V_{\text{tuple}} + V_{\text{alt}} + V_{\text{cont}}$$

$$V_{\text{int}} = \mathbb{Z} \qquad V_{\text{bool}} = \mathbb{B} \qquad V_{\text{fun}} = \left[V \underset{c}{\to} \left[V_{\text{cont}} \underset{c}{\to} V_* \right] \right]$$

$$V_{\text{tuple}} = \bigcup_{n \geq 0} V^n \qquad V_{\text{alt}} = \mathbb{N} \times V \qquad V_{\text{cont}} = \left[V \underset{c}{\to} V_* \right]$$

If we substitute the definition of V_{cont} in the definition of V_{fun}, we get

$$V_{\text{fun}} = \left[V \underset{c}{\to} \left[V \underset{c}{\to} V_* \right] \underset{c}{\to} V_* \right].$$

So, elements in V_{fun} are higher-order function. We would like to have a semantics that avoids using such higher-order functions. Such a semantics is called first-order.

(2) Before answering the question raised in (1), let me say a few words about why we are interested in such a first-order semantics. The first reason is a bit theoretical. It is that defining such a first-order semantics involves solving much simpler and easier recursive domain equations. In our original continuation semantics, we assumed that V is a solution of the following recursive (pre)domain equation:

$$V \simeq \left(\begin{array}{ll} & \mathbb{Z} + \mathbb{B} \\ + & \left[\begin{array}{r} \underline{\underline{V}} \underset{c}{\to} \left[\underline{V} \underset{c}{\to} \left(\underline{\underline{V}} + \text{error} + \text{typeerror} \right)_\perp \right] \\ \underset{c}{\to} \left(\underline{V} + \{\text{error}, \text{typeerror}\} \right)_\perp \end{array} \right] \\ + & \bigcup_n^\infty \underline{V}^n \\ + & \mathbb{N} \times \underline{V} \\ + & \left[\underline{\underline{V}} \underset{c}{\to} \left(\underline{V} + \text{error} + \text{typeerror} \right)_\perp \right] \end{array} \right)$$

Note that V appears on the both sides of $\to$. The occurrences of V underlined with two lines $\underline{\underline{V}}$ make this recursive predomain equation very difficult to solve. We should use the categorical fixed point theorem and the category of domains with

embeddings (which we covered before) to solve this equation. On the other hand, in the first-order semantics, we have a recursive predomain equation that is much easier to solve. It doesn't have those tricky recursive occurrences of $\hat{V}$ (a predomain being defined) that appear on the left argument side of $\rightarrow$.

The second reason is that this first-order continuation semantics becomes a theoretical basis or guide for a compiler for eager functional languages. The situation is analogous to the CPS transformation that we looked at. The transformation is derived from the continuation semantics. Similarly, from the first-order semantics, we are able to derive a program (or expression) transformation sometimes called defunctionalization, which gets rid of all higher-order functions. By the say, this kind of connection between (denotational) semantics and compilation should not be too surprising. In a sense, a denotational semantics is a compiler of programs into phrases in mathematics. If the compiler uses only very restricted phrases, the compiled phrases can be understood as instruction sequences in a computer.

(3) Let's define the first-order semantics. It is based on the observation that when we interpret an expression e in the continuation semantics, we do not use all functions, but specific kinds of functions. In a sense, the first-order semantics replaces V_{fun}, V_{cont}, and $E = V^{\text{var}}$ by three sets $\hat{V}_{\text{fun}}$, $\hat{V}_{\text{cont}}$, and $\hat{E}$, that consist of mathematical instructions. Then, it defines how to interpret those instructions.

We consider an eager functional language with integers and continuation values. Here are predomains and domains used in the first-order semantics.

$$\hat{V}_* = \left(\hat{V} + \{\text{error}, \text{typeerror}\}\right)_\perp$$

$$\hat{V} \underset{\psi}{\overset{\phi}{\rightleftarrows}} V_{\text{int}} + \hat{V}_{\text{fun}} + \hat{V}_{\text{cont}} \qquad V_{\text{int}} = \mathbb{Z}$$

$$\hat{V}_{\text{fun}} = \{\text{abstract}\} \times \langle var \rangle \times \langle exp \rangle \times \hat{E}$$

$\hat{V}_{\text{fun}}$: typical element $\ldots \langle \text{abstract}, v, e, \eta \rangle$. abstract indicates this tuple represents a lambda expression $\lambda v.e$ and an environment η for the free variables in the expression.

$$\begin{aligned}\hat{E} = &\{\text{initenv}\} \\ &\cup \{\text{extend}\} \times \langle var \rangle \times \hat{V} \times \hat{E} \\ &\cup \{\text{revenv}\} \times \hat{E} \times \langle var \rangle \times \langle var \rangle \times \langle exp \rangle\end{aligned}$$

initenv is the initial empty environment. $\langle \text{extend}, v, z, \eta \rangle$ is the environment obtained by extending η with the binding $v \mapsto z$. $\langle \text{revenv}, \eta, u, v, e \rangle$ is the environment obtained by extending η with the recursively defined u (i.e., $u = \lambda v.e$).

$$\begin{aligned}\hat{V}_{\text{cont}} = &\{\text{negate}\} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{add}_1, \text{div}_1, \text{mul}_1\} \times \langle exp \rangle \times \hat{E} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{add}_2, \text{div}_2, \text{mul}_2\} \times V_{\text{int}} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{app}_1\} \times \langle exp \rangle \times \hat{E} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{app}_2\} \times \hat{V}_{\text{fun}} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{ccc}\} \times \hat{V}_{\text{cont}} \\ &\cup \{\text{thw}\} \times \langle exp \rangle \times \hat{E} \\ &\cup \{\text{initcont}\}\end{aligned}$$

negate negates its input and call the continuation. The second and third cases are continuations for addition, division, and multiplication. The fourth and fifth cases are continuations for function application. Others are continuations for callcc, throw, and the initial continuation.

Note that elements of $\hat{V}_{\text{fun}}, \hat{V}_{\text{cont}}$, and $\hat{E}$ are not functions. Rather they are like instructions that denote certain functions. They are almost like programs.

The semantic function $[\![-]\!]$ has a slightly more complex definition. It is because the definition should now spell out how we can view elements of $\hat{V}_{\text{fun}}, \hat{V}_{\text{cont}}$, and $\hat{E}$ as appropriate functions. We define three more functions:

$$\begin{aligned}[\![-]\!] &\in \left[\langle exp \rangle \to \hat{E} \to \hat{V}_{\text{cont}} \to \hat{V}_* \right] \\ \text{cont} &\in \left[\hat{V}_{\text{cont}} \to \left[\hat{V} \to \hat{V}_* \right] \right] \\ \text{apply} &\in \left[\hat{V}_{\text{fun}} \to \left[\hat{V} \to \hat{V}_{\text{cont}} \to \hat{V}_* \right] \right] \\ \text{get} &\in \left[\hat{E} \to \left[\langle var \rangle \to \hat{V} \right] \right]\end{aligned}$$

Here cont, apply and get functions provide the meanings of elements (or records or instructions) in $\hat{V}_{\text{cont}}$, $\hat{V}_{\text{fun}}$, and $\hat{E}$. Whenever we need to use those elements by, say, look-up and function application, we use these three functions. These three functions and $[\![-]\!]$ are defined mutually recursively as follows:

$$\text{apply} \, \langle \text{abstract}, v, e, \eta \rangle \, a \, \kappa = [\![e]\!] \, \langle \text{extend}, v, a, \eta \rangle \, \kappa$$

$$\text{get} \, \langle \text{initenv} \rangle \;\; v = \psi \, \langle 0, 0 \rangle \quad \text{(initial value 0 assigned to } v\text{)}$$

$$\text{get} \, \langle \text{extend}, v, a, \eta \rangle \;\; w = \text{if } v = w \begin{cases} \text{then } a \\ \text{else get } \eta \; w \end{cases}$$

$$\begin{aligned}&\text{get} \, \langle \text{revenv}, \eta, v, u, e \rangle \;\; w \\ &\qquad = \text{if } v = w \begin{cases} \text{then } \psi \left\langle 1, \left\langle \text{abstract}, u, e, \langle \text{revenv}, \eta, v, u, e \rangle \right\rangle \right\rangle \\ \text{else get } \eta \; w \end{cases}\end{aligned}$$

$$\text{cont}\,\langle \text{negate}, \kappa\rangle\ a = \Big(\lambda i.\ \text{cont}\ \kappa\ \big(\psi\,\langle 0, -i\rangle\big)\Big)_{\text{int}}\ a$$

$$\text{cont}\,\langle \text{add}_1, e, \eta, \kappa\rangle\ a = \big(\lambda i.\ [\![e]\!]\,\eta\,\langle \text{add}_2, i, \kappa\rangle\big)_{\text{int}}\ a$$

$$\text{cont}\,\langle \text{add}_2, i, \kappa\rangle\ a = \left(\lambda i'.\ \text{cont}\ \kappa\ \Big(\psi\,\big\langle 0, i + i'\big\rangle\Big)\right)_{\text{int}}\ a$$

$$\text{cont}\,\langle \text{div}_2, i, \kappa\rangle\ a = \left(\lambda i'.\ \text{if}\ i' = 0\ \begin{cases}\text{then}\ \kappa\ \big(\psi\,\langle 1, \text{error}\rangle\big)\\ \text{else cont}\ \kappa\ \Big(\psi\,\big\langle 0, i \div i'\big\rangle\Big)\end{cases}\right)_{\text{int}}\ a$$

$$\text{cont}\,\langle \text{app}_1, e, \eta, \kappa\rangle\ a = \big(\lambda f.\ [\![e]\!]\,\eta\,\langle \text{app}_2, f, \kappa\rangle\big)_{\text{fun}}\ a$$

$$\text{cont}\,\langle \text{app}_2, f, \kappa\rangle\ a = \text{apply}\ f\ a\ \kappa$$

$$\text{cont}\,\langle \text{ccc}, \kappa\rangle\ a = \Big(\lambda f.\ \text{apply}\ f\ \big(\psi\,\langle 2, \kappa\rangle\big)\ \kappa\Big)_{\text{fun}}\ a$$

$$\text{cont}\,\langle \text{thw}, e, \eta\rangle\ a = \big(\lambda \kappa'.\ [\![e]\!]\,\eta\ \kappa'\big)_{\text{cont}}\ a$$

$$\text{cont}\,\langle \text{initcont}\rangle\ a = \psi\,\langle 0, a\rangle \quad \text{(0th component of } \big(V + \{\text{error}, \text{typeerror}\}\big)_{\perp}\text{)}$$

$\langle \text{mul}_1, e, \eta, \kappa\rangle$, $\langle \text{mul}_2, i, \kappa\rangle$ and $\langle \text{div}_1, e, \eta, \kappa\rangle$ are all interpreted similarly to $\langle \text{add}_1, e, \eta, \kappa\rangle$ and $\langle \text{add}_2, i, \kappa\rangle$.

$$[\![n]\!]\,\eta\ \kappa = \text{cont}\ \kappa\ \big(\psi\,\langle 0, n\rangle\big)$$

$$[\![-e]\!]\,\eta\ \kappa = [\![e]\!]\,\eta\ \ \langle \text{negate}, \kappa\rangle$$

$$[\![e_0 + e_1]\!]\,\eta\ \kappa = [\![e_0]\!]\,\eta\ \ \langle \text{add}_1, e_1, \eta, \kappa\rangle$$

$$[\![e_0 \div e_1]\!]\,\eta\ \kappa = [\![e_0]\!]\,\eta\ \ \langle \text{div}_1, e_1, \eta, \kappa\rangle$$

$$[\![e_0 \times e_1]\!]\,\eta\ \kappa = [\![e_0]\!]\,\eta\ \ \langle \text{mul}_1, e_1, \eta, \kappa\rangle$$

$$[\![v]\!]\,\eta\ \kappa = \text{cont}\ \kappa\,(\text{get}\ \eta\ v)$$

$$[\![e_0\ e_1]\!]\,\eta\ \kappa = [\![e_0]\!]\,\eta\ \ \langle \text{app}_1, e_1, \eta, \kappa\rangle$$

$$[\![\lambda v.e]\!]\,\eta\ \kappa = \text{cont}\ \kappa\ \Big(\psi\,\big\langle 1, \langle \text{abstract}, v, e, \eta\rangle\big\rangle\Big)$$

$$[\![\text{callcc}\ e]\!]\,\eta\ \kappa = [\![e]\!]\,\eta\ \ \langle \text{ccc}, \kappa\rangle$$

$$\big[\!\big[\text{throw}\ e\ e'\big]\!\big]\,\eta\ \kappa = [\![e]\!]\,\eta\ \ \big\langle \text{thw}, e', \eta\big\rangle$$

$$[\![\text{error}]\!]\,\eta\ \kappa = \langle 1, \text{error}\rangle$$

$$[\![\text{typeerror}]\!]\,\eta\ \kappa = \langle 1, \text{typeerror}\rangle$$

$$[\![\text{letrec}\ v_0 \equiv \lambda u_0.e_0\ \text{in}\ e_1]\!]\,\eta\ \kappa = [\![e_1]\!]\,\langle \text{revenv}, \eta, v_0, u_0, e_0\rangle\ \kappa$$

Note that this recursive definition is well-defined because of the following two reasons.

(i) get is defined inductively[♣] and doesn't depend on $[\![-]\!]$, apply and cont.

(ii) Since $\hat{V}_*$ is a domain and the function space $\left[P \underset{c}{\to} D \right]$ from a predomain P to a domain D is a domain, all of $D_1 = \left[\langle exp \rangle \to \hat{E} \to \hat{V}_{\text{cont}} \to \hat{V}_* \right]$, $D_2 = \left[\hat{V}_{\text{cont}} \to \left[\hat{V} \to \hat{V}_* \right] \right]$ and $D_3 = \left[\hat{V}_{\text{fun}} \to \left[\hat{V} \to \hat{V}_{\text{cont}} \to \hat{V}_* \right] \right]$ are domains. $[\![-]\!]$, cont and apply can be understood as a fixed point (in fact, the least fixed point) of some continuous function $F : D_1 \times D_2 \times D_3 \to D_1 \times D_2 \times D_3$. This function F is what the semantic definitions of $[\![-]\!]$, cont and apply determine.

(4) Let me mention two further points. First, the definition in the previous two pages doesn't use lambda in the mathematical meta language in a sense. Yes, you can see λ there. But those λ's are mainly for enabling the use of $(-)_\theta$ notation, which does runtime type checking. We could have used the unpacked definition of $(-)_\theta$ instead and avoided λ completely.

This lack of λ confirms that the semantics is first-order. Second, the predomain equation for V can be solved in the category of sets, i.e., without using domain theory. That is, we can define a set V s.t.

$$V = {}^{\blacklozenge}\mathbb{Z} + V_{\text{fun}} + V_{\text{cont}}$$

where V_{fun} and V_{cont} are defined as before.

(5) I tried to derive the program transformation from this first-order semantics. But I couldn't find a simple way to do so. Sorry guys.

Let me instead show you how one can derive a small-step evaluation relation (or more commonly called small-step operational semantics) from the first-order denotational semantics. The idea is to replace $=$ by a single evaluation step $\to$.

[♣] all recursive calls in the definition of get are over sub-environments.
[♦] equality

$$
\begin{aligned}
\langle n, \eta, \kappa\rangle &\to \langle \text{cont}, \kappa, n\rangle \\
\langle -e, \eta, \kappa\rangle &\to \left\langle e, \eta, \langle \text{negate}, \kappa\rangle\right\rangle \\
\left\langle e_0 \begin{smallmatrix}+\\ \div \\ \times\end{smallmatrix} e_1, \eta, \kappa\right\rangle &\to \left\langle e_0, \eta, \left\langle \begin{smallmatrix}\text{add}_1\\ \text{div}_1\\ \text{mul}_1\end{smallmatrix}, e_1, \eta, \kappa\right\rangle\right\rangle \\
\langle v, \eta, \kappa\rangle &\to \left\langle \text{cont}, \kappa, (\text{get } \eta\ v)\right\rangle \\
\langle e_0\ e_1, \eta, \kappa\rangle &\to \left\langle e_0, \eta, \langle \text{app}_1, e_1, \eta, \kappa\rangle\right\rangle \\
\langle \lambda v.e, \eta, \kappa\rangle &\to \left\langle \text{cont}, \kappa, \langle \text{abstract}, v, e, \eta\rangle\right\rangle \\
\langle \text{callcc } e, \eta, \kappa\rangle &\to \left\langle e, \eta, \langle \text{ccc}, \kappa\rangle\right\rangle \\
\left\langle \text{throw } e\ e', \eta, \kappa\right\rangle &\to \left\langle e, \eta, \left\langle \text{thw}, e', \eta\right\rangle\right\rangle \\
\langle \text{letrec } v_0 \equiv \lambda u_0.e_0 \text{ in } e, \eta, \kappa\rangle &\to \left\langle e, \langle \text{revenv}, \eta, v_0, u_0, e_0\rangle, \kappa\right\rangle \\
\left\langle \text{cont}, \langle \text{negate}, \kappa\rangle, a\right\rangle &\to \left\langle \kappa, \psi\,\langle 0, -a\rangle\right\rangle \quad (\text{if } a \in \mathbb{Z}) \\
\left\langle \text{cont}, \langle \text{add}_1, e, \eta, \kappa\rangle, a\right\rangle &\to \left\langle e, \eta, \langle \text{add}_2, a, \kappa\rangle\right\rangle \\
\left\langle \text{cont}, \langle \text{add}_2, a, \kappa\rangle, b\right\rangle &\to \langle \text{cont}, \kappa, a + b\rangle \quad (\text{if } a, b \in \mathbb{Z}) \\
&\qquad\qquad \text{mul}_1, \text{ mul}_2 \text{ and } \text{div}_1 \text{ are similar} \\
\left\langle \text{cont}, \langle \text{div}_2, a, \kappa\rangle, b\right\rangle &\to \langle \text{cont}, \kappa, a \div b\rangle \quad (\text{if } a, b \in \mathbb{Z} \text{ and } b \neq 0) \\
\left\langle \text{cont}, \langle \text{app}_1, e, \eta, \kappa\rangle, a\right\rangle &\to \left\langle e, \eta, \langle \text{app}_2, a, \kappa\rangle\right\rangle \\
\left\langle \text{cont}, \langle \text{app}_2, a, \kappa\rangle, b\right\rangle &\to \langle \text{apply}, a, b, \kappa\rangle \quad (\text{if } a \in \hat{V}_{\text{fun}}) \\
\left\langle \text{cont}, \langle \text{ccc}, \kappa\rangle, a\right\rangle &\to \langle \text{apply}, a, \kappa, \kappa\rangle \quad (\text{if } a \in \hat{V}_{\text{fun}}) \\
\left\langle \text{cont}, \langle \text{thw}, e, \eta\rangle, a\right\rangle &\to \langle e, \eta, a\rangle \quad (\text{if } a \in \hat{V}_{\text{cont}}) \\
\left\langle \text{cont}, \langle \text{initcont}\rangle, a\right\rangle &\to a \\
\left\langle \text{apply}, \langle \text{abstract}, v, e, \eta\rangle, a, \kappa\right\rangle &\to \left\langle e, \langle \text{extend}, v, a, \eta\rangle, \kappa\right\rangle
\end{aligned}
$$

We use definition of get for environments in $\hat{E}$.