CS520 Lecture 3
Simple Imperative Language

January 12, 2020

3.1 Motivation or objective

1. Most real-world PLs support computation by state update and that by function application. The former is often referred as imperative computation, and the latter as functional or applicative computation. Our goal is to study core of PL concepts and ideas for imperative computation.
2. Acutally, it is more appropriate to say that our aim is a formal & 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 1) some basic concepts and results of domain theory, 2) variable declaration and binding, 3) syntactic sugar, error handling and 4) 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 bit, and compare your answer with what I'll explain.

3.2 Syntax

1. Variables, and read and update of them . . . key concepts or operations for imperative computation.
2. Syntax supports these concepts and operations:

$\langle$intexp$\rangle$ ::= 0|1|. . . |$\langle$var$\rangle^1$|. . . − same as before

$\langle$boolexp$\rangle$ ::= $true|false|\ldots -$

almost the same as that of $\langle$assert$\rangle$.

The exception is that $\langle$boolexp$\rangle$ doesn't include quantifiers. Q. why?

$\langle$comm$\rangle$ ::= $\langle$var$\rangle$:= $\langle$intexp$\rangle^2|skip|\langle$comm$\rangle;\langle$comm$\rangle^3$

|if $\langle$boolexp$\rangle$ then $\langle$comm$\rangle$ else $\langle$comm$\rangle$

|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 an (multi-sorted) initial algebra for the signature determined by the grammar.
3. It is language for expressing a sequence of variable reads and variable updates.

3.3 Basic 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. $[\![-]\!] : \langle \text{comm} \rangle \to \ldots$ We want the following equation for loop holding to hold:

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

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

$$F \overset{informally}{\text{“ = ”}} [\![\text{if } b \text{ then } (c; -) \text{ else } skip]\!]$$

One objective of domain theory is to formalise good properties enjoyed by such program-implementable functions without going the low-level details of computability theory.
4. High-level meta heuristic behind domain theory

1. Consider a set together with some structure.
2. Use functions between such sets that respect the structures.
3. Why 1) and 2)? Because if done well, functions in 2) will always have fixed points.

5. Key definitions

Definition 1
A binary relation $\sqsubseteq$ on a sset S is a partial order if

1. *$x \sqsubseteq$ for all $x \in S$ (reflexivity)*
2. *for all $x, y, z \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (transivity)*
3. *for all $x, y \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.*

A set S with a partial order sqsubseteq is called partially ordered set of poset.

Definition 2
A chain in a poset $(S, \sqsubseteq)$ is a (countably) infinite sequence

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

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

Definition 3
A pre-domain is a poset $(S, \sqsubseteq)$ s.t. all chains have least upper bounds.
meaning : For every chain $\{x_n\}_{n \leq 0} \in S$, there exists $y = \sqcup_{n=0}^{\infty} x_n \in S$ s.t.
1) $x_n \sqsubseteq y$ for all $n \leq 0$
2) for any z, if $x_n \sqsubseteq z$ for every n, then $y \sqsubseteq z$.

Definition 4
A domain is a pre-domain $(S, \sqsubseteq)$ that has the least element, often denoted $\perp$.
meaning : for all $x \in S, \perp \sqsubseteq x$.

Definition 5
Let $(S_1, \sqsubseteq_1)$ *and* $(S_2, \sqsubseteq_2)$ *be pre-domains.*
A function $f \in [S_1 \to S_2]$ *is continuous if*
for every chain $\{x_n\}_{n\leq 0}$ *in S,* $f(\sqcup_{n=0}^{\infty} x_n)$ *is the least upper bound of* $\{f(x_n)\}_{n\leq 0}$
A function $f \in [S_1 \to S_2]$ *is monotone if for all* $x, y \in S_1$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
When S_1 *and* S_2 *are domains with least elements* $\perp_1$ *and* $\perp_2$, *we say that a function* $f \in [S_1 \to S_2]$ *is strict if* $f(\perp_1) = \perp_2$.

Exercise : Show that if f is continuous, it is monotone.
6. What's going on here? What are the intuituions behind these definitions?

1. $x \sqsubseteq y$ - intuitively means that y has more information than x or x and y have the same amount of info.

   (a) $\mathbb{Z}^{*,\omega} = \mathbb{Z}^* \cup \mathbb{Z}^\omega$ - finite or infinite sequence of integers.
   $x \sqsubseteq y$ iff x is an initial subsequence (or prefix) of y.

   $$\langle 3, 1, 4\rangle \sqsubseteq \langle 3, 1, 4, 1, 5\rangle$$

   $$\langle 3, 1, 4\rangle \not\sqsubseteq \langle 2, 1, 4, 1\rangle$$

   (b) $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ - lifted $\mathbb{Z}$

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

   (c) $(2^{\mathbb{Z}}, \subset)$

   (d) vertical domain of natural numbers -
   $$\begin{array}{c} \infty =: \top \\ \vdots \\ 1 \\ \perp = 0 \end{array}$$

2. The monotonicity means the preservation of the approximation relation.
   One intuition behind continuity of f is that in order to produce finite amound of information in its output, f consumes only finite amount information in its input.

   (a) $set \in [\mathbb{Z}^{*,\omega} \to 2^{\mathbb{Z}}]$

   $$set(\langle x_1, \ldots, x_n\rangle) = \{x_1, \ldots, x_n\}$$
   $$set(\langle x_1, \ldots, x_n, \ldots\rangle) = \{x_1, \ldots, x_n, \ldots\}$$

   If $A \subset set(s)$ and A is finite, then there is a finite prefix s_0 of s (i.e., $s_0 \sqsubseteq s$) s.t. $A \subset set(s_0)$
   Exercise : Show that if $f \in [\mathbb{Z}^{*,\omega} \to 2^{\mathbb{Z}}]$ is continuous, it satisfies above property. Also show that if $f \in [\mathbb{Z}^{*,\omega} \to 2^{\mathbb{Z}}]$ satisfies above property, then f is continuous.

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

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

Exercise : 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 $\{f(A_0) : A_0 \overset{\subset}{fin} A\}$.
Show that f is continuous iff it is finitely generated.

(c) 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 amound of information in a finite amount of time, nor produce output except at finite times."[4]

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

Proposition 1 (Least Fixed-Point Theorem)
If D is a domain and f is a continuous function from D to D,

$$x = \sqcup_{n=0}^{\infty} f^n \bot$$

is the least fixed-point of f. (That is, $f(x) = x$ and whenever $f(y) = y$, we have $x \sqsubseteq y$)

Proof. By the exercise before, f is monotone. Using induction and $\bot$ is being the least element, we can show that $\{f^n \bot\}_{n \leq 0}$ is a chain in D. Since D is a domain, the least upper bound $x := \sqcup_{n=0}^{\infty} f^n \bot$ exists. Furthermore, by the continuity of f,

$$f(x) = f(\sqcup_{n=0}^{\infty} f^n \bot) = \sqcup_{n=0}^{\infty} f^{n+1}(\bot) = \sqcup_{n=0}^{\infty} f^n(\bot) = x$$

$\therefore$ 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 sow that y is an upper bound of the chain $\{f^n \bot\}_{n \leq 0}$. $\therefore$ $x \sqsubseteq y$. □

8. When P, P' are predomains, we write $[P \to_c P']$ for the set of continuous functions. When $[P \to_c P']$[5] is given pointwise order $\sqsubseteq$

$$f \sqsubseteq g \text{ iff } f(x) \sqsubseteq_{P'} g(x) \text{ for all } x \in P, f, g \in [P \to_c P']$$

it becomes a predomain, where the limit of a chain $\{f_n\}_{n \leq 0}$ is defined pointwise $x \mapsto \sqcup_{n=0}^{\infty} f_n(x)$. Furthermore, if P' is a domain with the least element $\bot'$, then $[P \to_c P']$ 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 $[D \to_c D]$ to D:

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

[4]Domain theory attempty to capture this aspect of computation.

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

Lemma 1
Y_D is continuous.[6]

Proof. Exercise. □

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

1. $P_\perp := P \cup \{\perp\}$ for a predomain P.

$$x \sqsubseteq_{P_\perp} y \text{ iff } x = \perp \text{ or } x, y \in P \text{ and } x \sqsubseteq_P y \text{ for } x, y \in P_\perp$$

   Intuitively, we are adding the least element to P and converting P to a domain.

2. $i_\uparrow \in [P \to_c P_\perp]$, sometimes called unit, $i_\uparrow(x) = x$

3. for each $f \in [P \to_c P'_\perp]$,

$$f_{\shortparallel} \in [P_\perp \to_c P'_\perp]$$
$$f_{\shortparallel}(\perp) := \perp$$
$$f_{\shortparallel}(x) := f(x)$$

   sometimes called kleisli extension.

4. Why should we care about 2. and 3. ? Because they allow us to compose continuous function from P to $P'_\perp$

$$f \in [P \to_c P'_\perp] \text{ and } g \in [P' \to_c P''_\perp]$$
$$\Rightarrow (g_{\shortparallel} \circ f) \in [P \to_c P''_\perp]^7$$

3.4 Denotational Semantics of the simple imperative language

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

$$[\![-]\!]_{intexp} \in \langle \text{intexp} \rangle \to [\Sigma \to \mathbb{Z}]$$
$$[\![-]\!]_{boolexp} \in \langle \text{boolexp} \rangle \to [\Sigma \to \mathbb{B}]$$
$$[\![-]\!]_{comm} \in \langle \text{comm} \rangle \to [\Sigma \to_c \Sigma_\perp]$$

[6] This means that the very act of computing a fixed point of a given function is continuous.
[7] We can view $(-)_{\shortparallel} \circ (c)$ as a new composition operator $\circ'$

$$[\![x := e]\!]\sigma = [\sigma | x : [\![e]\!]\sigma]$$
$$[\![skip]\!]\sigma = \sigma$$
$$[\![c_1; c_2]\!]\sigma = [\![c_2]\!]_{\perp\!\!\perp}([\![c_1]\!]\sigma)$$
$$[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!]\sigma = \text{if } [\![b]\!]\sigma \text{ then } [\![c_1]\!]\sigma \text{ else } [\![c_2]\!]\sigma$$
$$[\![\text{while } b \text{ do } c]\!]\sigma = Y_{\Sigma_\perp}(F)$$
$$\text{where } F \in [\Sigma \to_c \Sigma_\perp] \to_c [\Sigma \to_c \Sigma_\perp]$$
$$Ffx\sigma := \text{if } [\![b]\!]\sigma \text{ then } (f_{\perp\!\!\perp} \circ [\![c]\!])\sigma \text{ else } \sigma$$

2. Why least 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 $F(f) = f$.

1. least fixed point - $[\![\text{while } true \text{ do } skip]\!]\sigma = \perp$
2. non-least fixed point - $[\![\text{while } true \text{ do } skip]\!]\sigma = \sigma$

$$F(f)\sigma = \text{if } [\![true]\!]\sigma \begin{cases} \text{then } (f_{\perp\!\!\perp} \circ [\![skip]\!])\sigma \\ \text{else } \sigma \end{cases} \tag{3.1}$$
$$= f(\sigma) \tag{3.2}$$

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

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

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 $[\![-]\!]_{intexp}$ is the most important step in defining the semantics. It also clarifies certain major design decisions of the target PL.

3.5 Variable declaration and substitution

1.

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

Construct that doesn't increase the expressivity of the language but enables the programmers to combat the complexity of software by introducity the idea of scope and local variables.
2.

$$[\![\text{newvar } v := e \text{ in } c]\!]\sigma = (\underbrace{(\lambda\sigma' \in \Sigma.[\sigma' | v : \sigma v])_{\perp\!\!\perp}}_{\text{restore the old value}}{}^{8} \circ \underbrace{[\![c]\!]}_{\text{run the body}})(\underbrace{[\sigma | v : [\![e]\!]\sigma]}_{initialize})$$

[8] means the function $\sigma' \mapsto [\sigma' | v : \sigma v]$

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 exprected properties like Coincidence [Prop2.6], Renaming [Prop2.8].

$FV(c)$ – free variables appearing in c p40

$FA(c)$ – free assigned variables appearing in c p41

Proposition 2 (Coincidence)

$$
\begin{aligned}
&(a)\sigma\omega = \sigma'\omega \textit{ for all } \omega \in FV(c)\\
&\quad \Rightarrow ([\![c]\!]\sigma = [\![c]\!]\sigma' = \bot) \textit{ or}\\
&\qquad ([\![c]\!]\sigma, [\![c]\!]\sigma' \in \Sigma \textit{ and } ([\![c]\!]\sigma)(\omega) = ([\![c]\!]\sigma'(\omega \textit{ for all } \omega \in FV(c)))\\
&(b)[\![c]\!]\sigma \neq \bot \Rightarrow ([\![c]\!]\sigma)(\omega) = \sigma\omega \textit{ for all } \omega \notin FA(c)
\end{aligned}
$$

Proposition 3 (Renaming)

$$
\begin{aligned}
&v_{new} \notin FV(c') - \{v\}\\
&\Rightarrow [\![\textit{newvar } v := e \textit{ in } c']\!]\sigma = [\![\textit{textnewvar} v_{new} := e \textit{ in } {}^{c'}/_{v \to v_{new}}]\!]\sigma
\end{aligned}
$$

3.6 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:

1. (for $v := e_0$ to $e_1 d_0 c$) := ($v := e_0$; while $v \leq e_1 d_0 (c; v := v + 1)$)
2. (for $v := e_0$ to $e_1 d_0 c$) := (newvar $v := e_0$ in while $v \leq e_1$ do $(c; v := v + 1)$)
3. (for $v := e_0$ to $e_1 d_0 c$) := (newvar $w := e_1$ in newvar $v := e_0$ in while $v \leq w$ do $(c; v := v + 1)$)
   where $w \neq v$ and $w \notin FV(e_0) \cup FV(c)$
4. 3. with the condition $v \notin FV(c)$

3. The for loop should be something easier to understand while. In this regard, 1 ¡ 2 ¡ 3 ¡ 4.

3.7 Arithmetic Errors

1. How should we deal with $x \div 0$ or underflow/overflow?
2. Two approaches

1. early stop with error

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

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

3.8 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,

$$[\![c_1]\!] = [\![c_2]\!] \Rightarrow c_1 \overset{=}{\text{our intuitive notion of equality defined separately}} c_2$$

This property is called soundness. Its converse is called full abstraction.

$$[\![c_1]\!] = [\![c_2]\!] \Leftarrow c_1 = c_2$$

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

$$c_1 \text{“ = ”} c_2 \text{ iff } \forall c \in \mathscr{C} \underline{\forall O \in \mathscr{O}}^{9} O(\underbrace{C[c_1]}_{\text{filling the hole of } C \text{ with } c_1}) = O(C[c_2])$$

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

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

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

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

[9] 1. Intuitively this condition says that under all use cases, the user cannot observe the difference between c_1 and c_2.
2. This is sometimes called observational equivalence.
3. Note that this is not a syntax-directed (or compositional) definition.