CS520 Lecture 5
Failure, Input-Output, and Continuation

March 20, 2020

5.1 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 executions.
2. We will study mathematical tools that allow us to study or analyze constructs for input-output and the fail operation, the simplist 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.

1. Recursively defined domains can be used to model computations with intermediate results (such as computations with outputs) and computations that may be stopped in the middle and be resumed later (such as computations with inputs).

2. Continuations may simplify semantic definitions by providing a canonical treatment of sequential composition operator.

5.2 Syntax of a programming language with failure & input-output

1.

$$\begin{aligned}\langle\text{comm}\rangle ::= & \overbrace{\ldots \mid \text{newvar } \langle\text{var}\rangle := \langle\text{intexp}\rangle \text{ in } \langle\text{comm}\rangle}^{\text{old stuff}} \\ & \mid \underbrace{\text{fail}}_{failure} \mid \underbrace{?\langle\text{var}\rangle}_{input} \mid \underbrace{!\langle\text{intexp}\rangle}_{output}\end{aligned}$$

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

2. <u>example</u>

$$x := 124; \text{while true do } (?y; \text{if } (y = 0) \text{ then fail else } (x := x/y; y; !x))$$

3. <u>exercises</u>

- write two interesting programs in this language.

- what should be the final state of the following program?

$$x := 124; y := 0; (\text{newvar } x := 3 \text{ in fail}); y := 2$$

5.3 Semantics

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

$$[\![-]\!] \in [\langle \text{comm} \rangle \to \underbrace{[A \to_C B]}_{continuous functions}]$$

what predomains or domains should we use for A and B?
2. A should be $\Sigma = [\langle \text{var} \rangle \to \mathbb{Z}]$, the set (or predomains) of states. (Recall that a set can be viewed as a predomain when it is given = as its order $\sqsubseteq$, which is called discrete order).
3. B is complex. Its elements describe computations that can be stopped and resument, and may output some integers in the middle. Formally,

$$B = \overbrace{\Omega \simeq (\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to_C \Omega))_\perp}^{\Omega \text{ satisfies a form of equation}}$$

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:

non-termination	$\cdots$	$\perp$
normal termination with a state σ	$\cdots$	$\sigma \in \Sigma$
abnormal termination with a state σ	$\cdots$	$\langle abort, \sigma \rangle$
output n and the rest of computation ω	$\cdots$	$\langle n, \omega \rangle \in \mathbb{Z} \times \Omega$
suspended computation g that waits for an input	$\cdots$	$g \in \mathbb{Z} \to_C \Omega$

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

$$P_0 + \ldots + P_{n-1} = \{\langle 0, x \rangle | x \in P_0\} \cup \ldots \cup \{\langle n-1, x \rangle | x \in P_{n-1}\}$$
$$\langle i, x \rangle \sqsubseteq \langle j, y \rangle \text{ iff } i = j \text{ and } x \sqsubseteq_i y$$
$$P_0 \times \ldots \times P_{n-1} = \{\langle x_0, \ldots, x_{n-1} \rangle | x_i \in P_i \text{ for all } i\}$$
$$\langle x_0, \ldots, x_{n-1} \rangle \sqsubseteq \langle y_0, \ldots, y_{n-1} \rangle \text{ iff } x_i \sqsubseteq_i y_i \text{ for all } i$$

1) exercise. Show that $P_0 + \ldots P_{n-1}$ and $P_0 \times P_{n-1}$ are pre-domains. Also, prove that $P_0 \times P_{n-1}$ is a domain if all of P_i's are domains.
Hint/Information
a. The least upper bound of a chain $\{\langle x_0^{(k)}, \ldots, x_{n-1}^{(k)} \rangle\}_k$ in $P_0 \times \ldots \times P_{n-1}$ can be computed componentwise.

$$\sqcup_{k=0}^{\infty} \langle x_0^{(k)}, \ldots, x_{n-1}^{(k)} \rangle = \langle \sqcup_{k=0}^{\infty} x_0^{(k)}, \ldots, \sqcup_{k=0}^{\infty} x_{n-1}^{(k)} \rangle$$

b. For every chain $\{z_k\}_k$ in $P_0 + \ldots + P_{n-1}$, there are some i and a chain $\{x_k\}_k$ in P_i s.t. $z_k = \langle i, x_k \rangle$.

2) THese predomain constructors correspond to the sum and product type constructors in programming languages.
3) For each case, we have a way to construct an element, and a way to destruct an element.
Constructors :

1. injection function

$$\begin{aligned} &i_k \in [P_k \to_C P_0 + \ldots + P_{n-1}] \\ &i_k(x) = \langle k, x \rangle \end{aligned}$$

2. For $f_0 \in [P \to_C P_0], \ldots, f_{n-1} \in [P \to_C P_{n-1}]$, the "target-tupling" function

$$\begin{aligned} &f_0 \otimes \ldots \otimes f_{n-1} \in [P \to_C P_0 \times \ldots \times P_{n-1}] \\ &(f_0 \otimes \ldots \otimes f_{n-1})(x) = \langle f_0(x), \ldots, f_{n-1}(x) \rangle \end{aligned}$$

Destructors :

1. For $f_0 \in [P_0 \to_C P], \ldots, f_{n-1} \in [P_{n-1} \to_C P]$, the "source-tupling" function

$$\begin{aligned} &f_0 \oplus \ldots \oplus f_{n-1} \in [P_0 + \ldots + P_{n-1}] \to_C P] \\ &(f_0 \oplus \ldots \oplus f_{n-1})(\langle i, x \rangle) = f_i(x) \end{aligned}$$

2. projection function

$$\begin{aligned} &\pi_k \in [P_0 \times \ldots \times P_{n-1} \to_C P] \\ &\pi_k \langle x_0, \ldots, x_{n-1} \rangle = x_k \end{aligned}$$

These constructors and destructors are mutually inveres in a sense Prop 5.1 and Prop 5.2 in the textbook express such inverse relationships.
4) The sum and product operators can be applied to continuous functions so as to build a new continuous functions. For instance, if

$$f_0 \in [P_0 \to_C P'_0], \ldots, f_{n-1} \in [P_{n-1} \to_C P''_{n-1}]$$

then we have the following two continuous functions

$$\begin{aligned} &(f_0 + \ldots + f_{n-1}) \in [P_0 + \ldots + P_{n-1} \to P'_0 + \ldots + P'_{n-1}] \\ &(f_0 + \ldots + f_{n-1})\langle i, x \rangle = \langle i, f_i(x) \rangle \\ &(f_0 \times \ldots \times f_{n-1}) \in [P_0 \times \ldots \times P_{n-1} \to P'_0 \times \ldots \times P'_{n-1}] \\ &(f_0 \times \ldots \times f_{n-1})\langle x_0, \ldots, x_{n-1} \rangle = \langle f_0(x_0), \ldots, f_{n-1}(x_{n-1}) \rangle \end{aligned}$$

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 foundation 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:

$$\Omega \simeq (\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to \Omega))_\perp$$
$$\hat{\Sigma} = \Sigma + \Sigma$$

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

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

such that

$$\phi \circ \psi = id \text{ and } \psi \circ \phi = id$$

Note that the RHS of $\simeq$ contains Ω, something that is being defined. It is a bit like Ω is defined in terms of Ω, i.e. recursively. Or we can say that Ω is a fixed point of some equations over domains.
1) Ω is the domain of possible outcomes. The isomorphism ϕ confirms that it consists of five kinds of elements correspondint to five different outcomes described earlier.
2) Ω is not just a solution of the recursive domain equation. It satisfies the following minimality condition (also called initiality).
For any domain D and any continuous function α

$$\alpha \in [(\hat{\Sigma} + (\mathbb{Z} \times D) + (\mathbb{Z} \to D))_\perp] \to_C D$$

there exists a unique continuous function β

$$\beta \in [\Omega \to_C D]$$

such that the following diagram commutes.

$$\begin{tikzcd}
(\hat{\Sigma} + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \to \Omega))_\perp \arrow[r, "(id_{\hat{\Sigma}}+(id_{\mathbb{Z}}\times\beta)+(\mathbb{Z}\to\beta))_\perp"] \arrow[d, "\phi"] & (\hat{\Sigma} + (\mathbb{Z} \times D) + ((Z) \to D))_\perp \arrow[d, "\alpha"] \\
\Omega \arrow[r, "\beta"] & D
\end{tikzcd}$$

$$(\mathbb{Z} \to \beta) \in [(\mathbb{Z} \to \Omega) \to_C (\mathbb{Z} \to D)$$
$$(\mathbb{Z} \to \beta)(g)(n) = \beta(g(n))$$

$$\forall f \in [D' \to_C D''], \text{ we have}$$
$$f_\perp \in [D'_\perp \to_C D''_\perp]$$
$$f_\perp(\alpha') = f(\alpha') \text{ for } \alpha' \in D'$$
$$f_\perp(\perp) = \perp$$

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.

3) Why does such Ω exist? Because the predomain constructors used in the RHS of $\backsimeq$ 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. Here, we use an analogous property on an operator that constructs a domain. Later we will study this in detail.
4) A few basic embedding operators

$$
\begin{aligned}
&i_\perp \in [\{\perp\} \to_C \Omega]\\
&i_\perp(\perp) = \psi(\perp_\Omega)\\
&i_{term} \in [\Sigma \to_C \Omega]\\
&i_{term}(\sigma) = \psi(\langle 0, \langle 0, \sigma\rangle\rangle) = \psi(\sigma)\\
&i_{abort} \in [\Sigma \to_C \Omega]\\
&i_{abort}(\sigma) = \phi(\langle 0, \langle 1, \sigma\rangle\rangle) = \psi(\langle abort, \sigma\rangle)\\
&i_{out} \in [\mathbb{Z} \times \Omega \to_C \Omega]\\
&i_{out}(n, \omega) = \psi(\langle 1, \langle n, \omega\rangle\rangle) = \psi(\langle n, \omega\rangle)\\
&i_{in} \in [[\mathbb{Z} \to \Omega] \to_C \Omega]\\
&i_{in}(g) = \psi(\langle 2, g\rangle) = \psi(g)
\end{aligned}
$$

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

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

How to define f_* and $h_\dagger$? Because 2), we can do it by case analysis

$$
\begin{aligned}
&f_*(i_\perp(\perp)) = i_\perp(\perp)\\
&f_*(i_{term}(\sigma)) = i_{term}(f(\sigma))\\
&f_*(i_{abort}(\sigma)) = i_{abort}(\sigma)\\
&f_*(i_{out}(n, \omega)) = i_{out}(n, f_*(\omega))\\
&f_*(i_{in}(g)) = i_{in}(\lambda n. f_*(g(n)))\\
&h_\dagger(i_\perp(\perp)) = i_\perp(\perp)\\
&h_\dagger(i_{term}(\sigma)) = i_{term}(h(\sigma))\\
&h_\dagger(i_{abort}(\sigma)) = i_{abort}(h(\sigma))\\
&h_\dagger(i_{out}(n, \omega)) = i_{out}(n, h_\dagger(\omega))\\
&h_\dagger(i_{in}(g)) = i_{in}(\lambda n. h_\dagger(g(n)))
\end{aligned}
$$

exercise. In both cases, we have well-defined f_* and $h_\dagger$ because of the minimality condition in 2). Find appropriate α and D.

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

$$i_{out}(3, i_{out}(4, \underline{\bot_\Omega}(= i_\bot(\bot)))) \sqsubseteq i_{out}(3, i_{out}(4, \underline{i_{out}(5, i_{term}(\sigma))}))$$
$$i_{in}(\lambda n. i_{out}(n+1, \underline{\bot_\Omega})) \sqsubseteq i_{in}(\lambda n. i_{out}(n+1, \underline{i_{in}(\lambda m. i_{out}(n+m, \bot_\Omega))}))$$

7. Interpretation of commands

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

exercises
1) Prove the following equations:

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

$$[\![fail; c]\!] = [\![fail]\!]$$

2) Does the following equation hold?

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

5.4 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/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:

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

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

$$[\![-]\!] \in [\langle\text{comm}\rangle \to \Sigma \to_C \Omega]$$

We have the extra part $(\Sigma \to_C \Omega) \to_C$, which expresses that $[\![-]\!]^{cont}$ takes an additional continuation parameter κ, compared with the (direct) semantics $[\![-]\!]$. Thus, when $\kappa \in [\Sigma \to_C \Omega], \sigma \in \Sigma$

$$\omega^1 = [\![c]\!]^{cont}(\kappa^2)(\sigma^3) \in \Omega$$

3. A good way to learn this continuation semantics is to complete the definition of $[\![-]\!]^{cont}$ based on the intuition that we discussed so far. So, hide the semantic clause in each case, and try to rediscover the clause for yourself.

$$[\![c]\!]^{cont} \in [[\Sigma \to_C \Omega] \to_C \Sigma \to_C \Omega]$$
$$[\![v := e]\!]^{cont}\kappa\sigma = \kappa([\sigma|v : [\![e]\!]\sigma])$$
$$[\![skip]\!]^{cont}\kappa\sigma = \kappa(\sigma)$$
$$[\![c_0; c_1]\!]^{cont}\kappa\sigma = [\![c_0]\!]^{cont}(\lambda\sigma'.[\![c_1]\!]^{cont}\kappa\sigma')\sigma$$
$$[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!]^{cont}\kappa\sigma = \text{ if } [\![b]\!]\sigma \text{ then } [\![c_0]\!]^{cont}\kappa\sigma]textelse[\![c_1]\!]^{cont}\kappa\sigma$$
$$[\![\text{while } b \text{ do } c_0]\!]^{cont}\kappa\sigma = (Y_{\Sigma\to_C\Omega}(F))(\sigma) = (\sqcup_{n=0}^{\infty}F(\bot))(\sigma)$$
$$\text{where} F \in ([\Sigma \to_C \Omega] \to_C [\Sigma \to_C \Omega])$$
$$F(\kappa')(\sigma') = \text{if } [\![b]\!]\sigma \text{ then } [\![c_0]\!]^{cont}\kappa'\sigma' \text{ else } \kappa(\sigma')$$

As usual, the most tricky part is the case for while. Note that we define the operator F on the domain of continuation, $[\Sigma \to_C \Omega]$, not on the domain of commands $[[\Sigma \to_C \Omega] \to_C \Sigma \to_C \Omega]$, and then we use the least fixed point of F, which denotes some continuation in $[\Sigma \to_C \Omega]$. In the (direct) semantics, we used the domain of commands.
4. Our definition is not complete. we should define $[\![?v]\!]^{cont}$ and $[\![!e]\!]^{cont}$ and $[\![\text{newvar } v := e \text{ in } c]\!]^{cont}$.

$$[\![!e]\!]^{cont}\kappa\sigma = i_{out}([\![e]\!]\sigma, \kappa(\sigma)) \qquad (\text{Recall } i_{out} \in [\mathbb{Z}\times\Omega \to_C \Omega])$$
$$[\![?v]\!]^{cont}\kappa\sigma = i_{in}(\lambda k \in \mathbb{Z}.\kappa([\sigma|v : k])) \qquad (\text{Recall } i_{in} \in [[\mathbb{Z} \to_C \Omega] \to_C \Omega])$$
$$[\![\text{newvar } v := e \text{ in } c]\!]^{cont}\kappa\sigma = [\![c]\!]^{cont}(\lambda\sigma'.\kappa([\sigma'|v : \sigma(v)]))[\sigma|v : [\![e]\!]\sigma]$$

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:

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

exercise Prove the above equation.
6. So far we didn't consider the command fail. Now it is time to stop ignoring it. Can we add a new semantics clause for $[\![fail]\!]^{cont}$? what about the following?

$$[\![fail]\!]^{cont}\kappa\sigma = i_{abort}(\sigma) \qquad (\text{Recall } i_{abort} \in [\Sigma \to_C \Omega])$$

[1] result of running c and then κ on the state σ
[2] computation to be done after c
[3] initial state

Unfortunately, this simple definition doesn't work.

$$[\![\text{newvar } x := 1 \text{ in } fail]\!]^{cont}\kappa\sigma \qquad \text{(where } \sigma(x) = 0)$$
$$= [\![fail]\!]^{cont}(\lambda\sigma'.\kappa([\sigma'|x : \sigma(x)]))[\sigma|x : 1] \qquad = [\sigma|x : 1] \neq \sigma$$

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

$$[\![-]\!]_2^{cont} \in [\langle\text{comm}\rangle \to [\Sigma \to_C \Omega]^4 \to_C [\Sigma \to_C \Omega]^5 \to_C \Sigma \to_C \Omega]$$
$$[\![v := e]\!]_2^{cont}\kappa_t\kappa_f\sigma = \kappa_t([\sigma|v : [\![e]\!]\sigma])$$
$$[\![skip]\!]_2^{cont}\kappa_t\kappa_f\sigma = \kappa_t(\sigma)$$
$$[\![c_1; c_2]\!]_2^{cont}\kappa_t\kappa_f\sigma = [\![c_1]\!]_2^{cont}([\![c_2]\!]_2^{cont}\kappa_t\kappa_f)\kappa_f\sigma$$
$$[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!]_2^{cont}\kappa_t\kappa_f\sigma = \text{ if } [\![b]\!]\sigma \text{ then } [\![c_0]\!]_2^{cont}\kappa_t\kappa_f\sigma \text{ else } [\![c_1]\!]_2^{cont}\kappa_t\kappa_f\sigma$$
$$[\![\text{while } b \text{ do } c]\!]_2^{cont}\kappa_t\kappa_f\sigma = (Y_{\Sigma\to_C\Omega}(F))(\sigma) = (\sqcup_{n=0}^{\infty}F^n(\bot))(\sigma)$$
$$\text{where } F(\kappa)(\sigma') = \text{if } [\![b]\!]\sigma' \text{ then } [\![c]\!]_2^{cont}\kappa\kappa_f\sigma' \text{ else } \kappa_t(\sigma')$$
$$[\![\text{newvar } v := e \text{ in } c]\!]_2^{cont}\kappa_t\kappa_f\sigma =$$
$$[\![c]\!]_2^{cont}(\lambda\sigma'.\kappa_t([\sigma'|v : \sigma(v)]))(\lambda\sigma'.\kappa_f([\sigma'|v : \sigma(v)]))[\sigma|v : [\![e]\!]\sigma]$$
$$[\![fail]\!]_2^{cont}\kappa_t\kappa_f\sigma = \kappa_f(\sigma)$$
$$[\![!e]\!]_2^{cont}\kappa_t\kappa_f\sigma = i_{out}([\![e]\!]\sigma, \kappa_t(\sigma))$$
$$[\![?v]\!]_2^{cont}\kappa_t\kappa_f\sigma = i_{in}(\lambda n.\kappa_t([\sigma|v : n]))$$