Solutions

1 Domain theory

a. (10 pts.) Assume that $f, g : D \to D$ are continuous functions on a pointed cpo D with $f(\bot) = g(\bot)$ and $f \circ g = g \circ f$. Prove that $\mathsf{fix}\ f = \mathsf{fix}\ g$.

Lemma: $f^n(\bot) = g^n(\bot)$ (by induction on n)

$$\begin{array}{rcll} f^n(\bot) & = & f^{n-1}(f(\bot)) & \\ & = & f^{n-1}(g(\bot)) & \\ & = & g(f^{n-1}(\bot)) & \textit{(by commutativity)} \\ & = & g(g^{n-1}(\bot)) & \textit{(induction hypothesis)} \\ & = & g^n(\bot)) & \end{array}$$

Since $f^n(\bot) = g^n(\bot)$ for all n, $\bigsqcup f^n(\bot) = \bigsqcup g^n(\bot)$.

b. (10 pts.) Given partial orders $D_1, \ldots, D_n$ with respective bottom elements $\bot_1, \ldots, \bot_n$, the *smash product* $D_1 \otimes \cdots \otimes D_n$ is a partial order formed by taking tuples containing only non-bottom elements from the D_i plus a single new bottom element. Another way to think of it is as the ordinary product construction, but where all tuples containing any bottom component have been identified. Formally,

$$(D_1, \sqsubseteq_1) \otimes \cdots \otimes (D_n, \sqsubseteq_n) = (D, \sqsubseteq)$$

where

$$D = \{\langle d_1, \ldots, d_n\rangle \mid d_i \in D_i \wedge d_i \neq \bot_i \ ^{i \in 1..n}\} \cup \{\bot\}$$

and the relation $\sqsubseteq$ is the least partial order relation containing at least the relationships defined by the following two rules:

$$\frac{d_i \sqsubseteq_i d_i' \ ^{i \in 1..n}}{\langle d_1, \ldots, d_n\rangle \sqsubseteq \langle d_1', \ldots, d_n'\rangle}$$

$$\frac{}{\bot \sqsubseteq \langle d_1, \ldots, d_n\rangle}$$

Prove that the smash product of pointed cpos $D_1, \ldots, D_n$ is a pointed cpo.

First, note that the ordering relation defined on the smash product is a partial order (once all the reflexive relationships are added). The only non-trivial kind of ω-chain has the form

$$\ldots \sqsubseteq \langle d_1', \ldots, d_n'\rangle \sqsubseteq \langle d_1'', \ldots, d_n''\rangle \sqsubseteq \langle d_1''', \ldots, d_n'''\rangle \ldots$$

where $d_i' \sqsubseteq d_i'' \sqsubseteq d_i''' \ldots$

Any upper bound of such a chain has the form $\langle d_1, \ldots, d_n\rangle$ where d_i is an upper bound of d_i', d_i'', etc. Therefore, the tuple whose components are the least upper bounds componentwise is no larger than any other upper bound and must be the least upper bound itself. This argument is of course exactly the same as the argument for an ordinary product of cpos, which makes sense because the smash product is isomorphic to the lift of the product of the cpos minus their bottom elements.

Finally, D is a pointed CPO because it has a bottom element, $\bot$.

2 Operational semantics for concurrent processes

The new concurrent programming language LIMPA extends IMP with support for multiple executing processes, each with their own state, and interprocess communication primitives. The IMP language is extended with the following new syntax:

$$\begin{array}{rcl} c & ::= & \ldots \mid \mathsf{fork}\ c \mid \mathsf{send}\ p,\ a \mid X := \mathsf{recv}\ P \\ P & ::= & \{p_1, \ldots, p_n\} \end{array}$$

The fork command executes the command c in a new process whose state is initially the same as that of the process executing the fork. The send command sends the value of the expression a to the port p. Ports are assumed to come from some countable set of port identifiers whose precise form is unimportant. The recv command allows a process to read non-deterministically from any of a set of ports P. (The ability to read from a set of ports allows a receiver to multiplex multiple senders.) Both send and recv are blocking communication primitives: the sending process waits at a send until another process executes a matching recv, and vice versa. A given message is only delivered to a single process.

For example, the following program ping-pongs an integer counter back and forth between two processes, such that one process only sees the even integers and the other the odd ones. The main process forks two processes and then terminates, but the program executes until all processes are done.

```
x := 0;
fork (while x < 1000 do (send p1, x+1; x := recv {p1}));
fork (while x < 1000 do (x := recv {p1}; send p1, x+1))
```

The LIMPA designers need your ingenuity in designing the small-step operational semantics for the language. They have managed to decide on a configuration for the semantics; a configuration t is defined as:

$$t ::= c, \sigma \mid t_1 \square t_2$$

A configuration is essentially an unordered collection of process configurations ($\langle c, \sigma \rangle$ pairs) and can be written as $c_1, \sigma_1 \square \ldots \square c_n, \sigma_n$. Note that the symbol $\square$ is not part of the LIMPA language syntax! To allow the list of processes to be reordered and reassociated arbitrarily without destroying or creating processes, they have added the following rules already:

$$\begin{array}{rcl} t_1 \square t_2 & \rightarrow & t_2 \square t_1 \\ t_1 \square (t_2 \square t_3) & \rightarrow & (t_1 \square t_2) \square t_3 \\ (t_1 \square t_2) \square t_3 & \rightarrow & t_1 \square (t_2 \square t_3) \end{array}$$

The LIMPA designers understand IMP pretty well, but they are having trouble defining the small-step rules for fork, send, and recv.

a. (3 pts.) Define what configurations you would like to consider final in LIMPA. What kinds of stuck configurations, if any, exist?

One reasonable choice for a final configuration is a configuration of the form skip, σ.

Because send *and* recv *can block waiting for an appropriate process to exchange information with, a configuration* $c_1, \sigma_1 \square \ldots \square c_n, \sigma_n$ *is stuck if each of the commands* c_i *has the form* $\mathsf{send}\ p_i, n$ *or* $\mathsf{send}\ p_i, n; c$ *or* $X := \mathsf{recv}\ P_i$ *or* $X := \mathsf{recv}\ P_i; c$*, and none of the ports* p_i *is a member of any of the sets* P_i.

b. (5 pts.) Assuming $p \in P$, what should the configuration $\mathsf{send}\ p\ n,\ \sigma_1 \square X := \mathsf{recv}\ P,\ \sigma_2$ step to in order that progress is made and the message is transmitted to exactly one process?

$$\mathsf{skip}, \sigma_1 \square \mathsf{skip}, \sigma_2[X \mapsto n]$$

How about $\mathsf{send}\ p\ n; c_1,\ \ \sigma_1 \ \square\ X := \mathsf{recv}\ P; c_2,\ \ \sigma_2$, where c_1 and c_2 are arbitrary commands?

$$\mathsf{skip}; c_1, \sigma_1 \ \square\ \mathsf{skip}; c_2, \sigma_2[X \mapsto n]$$

or

$$c_1, \sigma_1 \ \square\ c_2, \sigma_2[X \mapsto n]$$

To write the rules for more general uses of send, recv and the existing IMP commands, a more general technique using evaluation contexts is needed. First of all, the designers would like to be able to express most of the semantics by building on the IMP SOS with the following rule, in which the relation in the premise is the original IMP small-step evaluation relation, and the function T is a suitably defined evaluation context.

$$\frac{\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{T[c, \sigma] \rightarrow T[c', \sigma']}$$

c. (5 pts.) Write a BNF definition of the evaluation context T. It may be useful to define more than one kind of evaluation context.

$$\begin{array}{rcl} T & ::= & [\cdot] \mid T \ \square\ t \\ C & ::= & [\cdot] \mid C; c \end{array}$$

d. (5 pts.) Use your evaluation context T to write a small-step rule for send and recv.

$$\frac{p \in P}{T[C_1[\mathsf{send}\ p\ n], \sigma_1 \ \square\ C_2[X := \mathsf{recv}\ P], \sigma_2] \rightarrow T[C_1[\mathsf{skip}], \sigma \ \square\ C_2[\mathsf{skip}], \sigma_2[X \mapsto n]]}$$

e. (5 pts.) Similarly, write a small-step rule for fork.

$$\frac{}{T[C[\mathsf{fork}\ c], \sigma] \rightarrow T[C[\mathsf{skip}], \sigma] \ \square\ c, \sigma}$$

f. (2 pts.) What other rules are needed to complete the LIMPA SOS?

For termination:

$$\frac{}{\mathsf{skip}, \sigma \ \square\ t \rightarrow t}$$

For evaluation of send*:*

$$\frac{\langle a, \sigma\rangle \rightarrow \langle a', \sigma\rangle}{T[C[\mathsf{send}\ p\ a, \sigma]] \rightarrow T[C[\mathsf{send}\ p\ a', \sigma]]}$$

3 A variation on continuation-passing style

In class we talked about CPS conversion from a call-by-value lambda calculus to a simpler CPS language in which the bodies of abstractions could only be "statements". However, the application expression in the language is perhaps still more complex than desirable if using CPS to represent low-level code. Consider the following CPS language in which expressions can only be bound to variables:

$$\begin{array}{rcl} e & ::= & \mathsf{fn}(x_1, x_2)\ s \mid \mathsf{halt} \\ s & ::= & x_1(x_2, x_3) \mid \mathsf{let}\ x = e\ \mathsf{in}\ s \end{array}$$

For simplicity, only two-argument functions and applications are present in this language. When we write down the SOS for this language, we encounter the small difficulty that in the substitution-based model we have been using, terms that are not legal in the source language will appear because of substitution. For the purposes of the operational semantics, we have a language similar to that described in class:

$$\begin{array}{rcl} e & ::= & \mathsf{fn}(x_1, x_2)\ s \mid x \mid \mathsf{halt} \\ s & ::= & e_1(e_2, e_3) \mid \mathsf{let}\ x = e\ \mathsf{in}\ s \end{array}$$

with the usual evaluation rules

$$\begin{array}{rcl} (\mathsf{fn}(x_1, x_2)\ s)(e_1, e_2) & \rightarrow & s\{e_1/x_1, e_2/x_2\} \\ \mathsf{let}\ x = e\ \mathsf{in}\ s & \rightarrow & s\{e/x\} \end{array}$$

However, we will consider a program well-formed only if it is closed and conforms to the first syntax above.

a. (10 pts.) Provide a definitional semantics for the call-by-value lambda calculus in terms of this more primitive CPS language.

Here is a from the lambda calculus to the simple CPS language. Note that the continuation k that is the second argument to $\mathcal{D}$ is always a variable name.

$$\begin{array}{rcll} \mathcal{T}[\![e]\!] & = & \mathsf{let}\ h = \mathsf{halt}\ \mathsf{in}\ \mathcal{D}[\![e]\!]h & \\ \mathcal{D}[\![x]\!]k & = & k(x, x) & \text{(The second } x \text{ is a dummy)} \\ \mathcal{D}[\![\lambda\ x\ e]\!]k & = & \mathsf{let}\ f = (\mathsf{fn}(x, k')\ \mathcal{D}[\![e]\!]k')\ \mathsf{in}\ k(f, f) & \text{(where } f, k' \notin FV(e) \cup \{k\}) \\ \mathcal{D}[\![e_1\ e_2]\!]k & = & \mathsf{let}\ k_1 = (\mathsf{fn}(f, d) & \\ & & \quad \mathsf{let}\ k_2 = (\mathsf{fn}(v, d)\ f(v, k))\ \mathsf{in}\ \mathcal{D}[\![e_2]\!]k_2 & \\ & &)\ \mathsf{in}\ \mathcal{D}[\![e_1]\!]k_1 & \text{(where } k_1, k_2 \notin FV(e_1) \cup FV(e_2) \cup \{k\}) \end{array}$$

b. (10 pts.) Expanding the language syntax in order to write the operational semantics is somewhat annoying. An alternative is to use an *explicit substitution model*, in which the substitutions in force (essentially, the current set of variable bindings) are represented explicitly. In this approach, the operational semantics configuration takes the form (γ, s), where γ is a map from variables to expressions e. Any free variables in s must necessarily be mapped by γ. The final configuration of the program has the form $(\gamma, x_0\ x_1\ x_2)$ and we will consider x_1 substituted by γ to be the result of the program.

Write the evaluation rules for function application and let in the explicit substitution semantics. (Hint: the environments we have been using in denotational semantics are essentially explicit substitutions.)

The problem statement was somewhat misleading because we need the environment γ to be a map from variables to closures *to get the right semantics. A closure is a pair of an environment and an expression (γ, e) such that γ tells how to interpret all free variables in e. With γ of this form, we obtain the following rules in which, as desired, we never use the substitution operator, and hence never obtain any expressions not in the language:*

$$\frac{}{(\gamma, \mathsf{let}\ x = \mathsf{halt}\ \mathsf{in}\ s) \rightarrow (\gamma[x \mapsto \mathsf{halt}], s)}$$

$$\frac{}{(\gamma, \mathsf{let}\ x = e\ \mathsf{in}\ s) \rightarrow (\gamma[x \mapsto (\gamma, e)], s)} \qquad (e \neq \mathsf{halt})$$

$$\frac{\gamma(x_0) = (\gamma', (\mathsf{fn}(x_1', x_2')s))}{(\gamma, x_0(x_1, x_2)) \rightarrow (\gamma'[x_1' \mapsto \gamma(x_1), x_2' \mapsto \gamma(x_2)], s)}$$

The problem statement understandably tempted many of you into writing semantics similar to the following:

$$\frac{}{(\gamma, \mathsf{let}\ x = e\ \mathsf{in}\ s) \to (\gamma[x \mapsto e], s)}$$

$$\frac{\gamma(x_0) = (\mathsf{fn}(x_1', x_2')\ s)}{(\gamma, x_0(x_1, x_2)) \to (\gamma[x_1' \mapsto \gamma(x_1), x_2' \mapsto \gamma(x_2)], s)}$$

Among the problems with this semantics is that the free variables in a function body are interpreted with respect to the environment at the call site, not with respect to the lexical environment: it is dynamically scoped. Because of the confusion arising from the problem statement, we were lenient in grading this problem.

c. (5 pts.) We'd like to determine that both the explicit substitution semantics and the usual semantics given earlier agree with each other. How would you formally express the statement that these semantics agree when a program terminates in both semantics? (You need not prove this statement.)

Suppose we define $\hat{\gamma}(s)$ and $\hat{\gamma}(e)$ to mean the substitution of all free variables in s and e respectively according to γ. Let γ_0 be the empty substitution. Then the two semantics agree on terminating computations if:

$$s \to^* \mathsf{halt}(e_1, e_2) \iff \exists \gamma', s' \ .\ (\gamma_0, s) \to^* (\gamma', s') \wedge \hat{\gamma}'(s') = \mathsf{halt}(e_1, e_2)$$

This is all we were looking for in this problem, but we can define the functions $\hat{\gamma}(s)$ and $\hat{\gamma}(e)$ in terms of γ as follows by induction on the structure of γ (viewed syntactically), s and e:

$$\begin{array}{rcll}
\hat{\gamma}(x) & = & x & \text{if } x \notin dom(\gamma) \\
\hat{\gamma}(x) & = & \hat{\gamma}'(e) & \text{if } \gamma(x) = (\gamma', e) \\
\hat{\gamma}(x) & = & \mathsf{halt} & \text{if } \gamma(x) = \mathsf{halt} \\
\hat{\gamma}(x_0(x_1, x_2)) & = & \hat{\gamma}(x_0)(\hat{\gamma}(x_1), \hat{\gamma}(x_2)) & \\
\hat{\gamma}(\mathsf{let}\ x = e\ \mathsf{in}\ s) & = & \mathsf{let}\ x = \hat{\gamma}(e)\ \mathsf{in}\ \hat{\gamma}'(s) & (\gamma' = \gamma \text{ without its mapping for } x, \text{ if any}) \\
\hat{\gamma}(\mathsf{halt}) & = & \mathsf{halt} & \\
\hat{\gamma}(\mathsf{fn}(x_1, x_2)\ s) & = & \mathsf{fn}(x_1, x_2)\ \hat{\gamma}'(s) & (\gamma' = \gamma \text{ without mappings for } x_1, x_2)
\end{array}$$

4 Denotational semantics and continuations

In a sense, continuations are the goto construct of functional languages. In this problem, we will look at this correspondence more carefully by defining a continuation-passing-style denotational semantics for a simple assembly language with labels and conditional jumps.

Consider the following simple assembly language:

$$\begin{array}{rcl}
L & \in & \mathit{Label} \\
X, Y & \in & \mathit{Loc} \\
n & \in & \mathbb{Z} \\
c & ::= & \mathsf{load}\ n\ X \mid \mathsf{move}\ X\ Y \mid \mathsf{inc}\ X \mid \mathsf{dec}\ X \mid \mathsf{label}\ L \mid \mathsf{jz}\ X\ L \mid c_1\ ;\ c_2
\end{array}$$

A program c is a sequence of commands made up of constant loads, assignments (the destination is second location), increments and decrements, labels, and conditional jumps on zero. Control "falls through" to the next command in the sequence if a conditional branch is not taken. A program terminates after executing the final command in the sequence. If the final command in the sequence is a conditional jump, the program terminates when the conditional branch is not taken.

For example, here is a program which computes the sum of the numbers from 1 to 5, storing the result in location S:

$$
\begin{array}{l}
\mathsf{load}\ 0\ Z\ ; \\
\mathsf{load}\ 5\ N\ ; \\
\mathsf{load}\ 0\ S\ ; \\
\mathsf{label}\ L_1\ ; \\
\mathsf{jz}\ N\ L_4\ ; \\
\mathsf{move}\ N\ M\ ; \\
\mathsf{label}\ L_2\ ; \\
\mathsf{jz}\ M\ L_3\ ; \\
\mathsf{inc}\ S\ ; \\
\mathsf{dec}\ M\ ; \\
\mathsf{jz}\ Z\ L_2\ ; \\
\mathsf{label}\ L_3\ ; \\
\mathsf{dec}\ N\ ; \\
\mathsf{jz}\ Z\ L_1\ ; \\
\mathsf{label}\ L_4
\end{array}
$$

Note that the target of a conditional jump can occur before or after the jump. In addition, we require all assembly language programs to satisfy a well-formedness criteria. This criteria requires

1. all labels referenced as the target of a conditional jump must have a defining occurrence in a label command
2. no label can have more than one defining occurrence in a label command

The denotation $\mathcal{A}[\![c]\!]$ of a program will be defined as follows:

$$
\begin{array}{rcl}
\mathcal{A}[\![c]\!] & = & (\pi_2\ (\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle \in \mathit{Jump} \times \mathit{Cont}.\ \mathcal{C}[\![c]\!]\langle\iota,\kappa_0\rangle))))\ \sigma_0 \\
\kappa_0 & = & \lambda\sigma \in \mathit{Store}.\ \lfloor\sigma\rfloor \\
\sigma_0 & = & \lambda X \in \mathit{Loc}.\ 0
\end{array}
$$

The idea is that $\iota \in \mathit{Jump}$ maps labels to continuations; i.e., $(\iota\ L)$ is the continuation that continues from label L. Consider ι as a *jump table* recording the address at which execution to resume after a jump. A fixed-point is necessary because defining occurrences of a label can occur before or after a conditional jump to that label.

$\mathcal{C}[\![c]\!]$ takes a pair $\langle\iota,\kappa\rangle$ where ι is an approximation of the final jump table and κ is the continuation prepared to accept the result of evaluating c. $\mathcal{C}[\![c]\!]$ returns a pair $\langle\iota',\kappa'\rangle$ where ι' is a better approximation of the final jump table and κ' is the continuation which accepts a store σ, evaluates c using ι in σ, and passes the resulting store σ' to κ.

Note that

$$\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle.\ \mathcal{C}[\![c]\!]\langle\iota,\kappa_0\rangle) = \langle\iota_f,\kappa_f\rangle$$

such that $\langle\iota_f,\kappa_f\rangle = \mathcal{C}[\![c]\!]\langle\iota_f,\kappa_0\rangle$. Therefore, κ_f is the continuation which accepts a store σ, evaluates c using ι_f in σ, and passes the resulting store σ' to κ_0. Further, ι_f corresponds to the final jump table, that is, the jump table with perfect information mapping labels to continuations. Projecting out κ_f and applying it to the initial store σ_0 yields the denotation for the entire program.

a. (6 pts.) Complete the domain equations for the semantics for the assembly language.

$$
\begin{aligned}
Jump &= Label \to Cont \\
Cont &= Store \to Answer \\
Store &= Loc \to \mathbb{Z} \\
Answer &= Store_\perp \\
\mathcal{A}[\![c]\!] &\in Answer \\
\mathcal{C}[\![c]\!] &\in Jump \times Cont \to Jump \times Cont
\end{aligned}
$$

b. (10 pts.) Complete the following cases for $\mathcal{C}$.

$$
\begin{aligned}
\mathcal{C}[\![\mathsf{load}\ n\ X]\!]\langle \iota, \kappa\rangle &= \langle \iota, \lambda\sigma \in Store.\ \kappa\ \sigma[X \mapsto n]\rangle \\
\mathcal{C}[\![\mathsf{move}\ X\ Y]\!]\langle \iota, \kappa\rangle &= \langle \iota, \lambda\sigma \in Store.\ \kappa\ \sigma[Y \mapsto \sigma(X)]\rangle \\
\mathcal{C}[\![\mathsf{inc}\ X]\!]\langle \iota, \kappa\rangle &= \langle \iota, \lambda\sigma \in Store.\ \kappa\ \sigma[X \mapsto \sigma(X)+1]\rangle \\
\mathcal{C}[\![\mathsf{dec}\ X]\!]\langle \iota, \kappa\rangle &= \langle \iota, \lambda\sigma \in Store.\ \kappa\ \sigma[X \mapsto \sigma(X)-1]\rangle \\
\mathcal{C}[\![\mathsf{label}\ L]\!]\langle \iota, \kappa\rangle &= \langle \iota[L \mapsto \kappa], \kappa\rangle \\
\mathcal{C}[\![\mathsf{jz}\ X\ L]\!]\langle \iota, \kappa\rangle &= \langle \iota, \lambda\sigma \in Store.\ \mathsf{if}\ \sigma(X) = 0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa\ \sigma\rangle \\
\mathcal{C}[\![c_1\ ;\ c_2]\!]\langle \iota, \kappa\rangle &= \mathcal{C}[\![c_1]\!]\ (\mathcal{C}[\![c_2]\!]\langle \iota, \kappa\rangle)
\end{aligned}
$$

c. (9 pts. total) Compute the denotations of the following programs. Be sure to show the key steps in your reasoning.

i. (3 pt.) $\mathcal{A}[\![\mathsf{inc}\ X]\!]$

$$
\begin{aligned}
&\mathcal{C}[\![\mathsf{inc}\ X]\!]\langle \iota, \kappa_0\rangle \\
&\quad = \langle \iota, \lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1]\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \mathcal{C}[\![\mathsf{inc}\ X]\!]\langle \iota, \kappa_0\rangle) \\
&\quad = \mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \langle \iota, \lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1]\rangle) \\
&\quad = \langle \perp_{Jump}, \lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1]\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{A}[\![\mathsf{inc}\ X]\!] \\
&\quad = (\pi_2\ (\mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \mathcal{C}[\![\mathsf{inc}\ X]\!]\langle \iota, \kappa_0\rangle)))\ \sigma_0 \\
&\quad = (\pi_2\ \langle \perp_{Jump}, \lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1]\rangle)\ \sigma_0 \\
&\quad = (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma_0 \\
&\quad = \kappa_0\ \sigma_0[X \mapsto \sigma_0(X)+1] \\
&\quad = \kappa_0\ \sigma_0[X \mapsto 1] \\
&\quad = \lfloor \sigma_0[X \mapsto 1] \rfloor
\end{aligned}
$$

ii. (3 pts.) $\mathcal{A}[\![\mathsf{jz}\ Z\ L\ ;\ \mathsf{inc}\ X\ ;\ \mathsf{label}\ L]\!]$

$$
\begin{array}{l}
\mathcal{C}[\![\mathsf{jz}\ Z\ L\ ;\ \mathsf{inc}\ X\ ;\ \mathsf{label}\ L]\!]\langle \iota, \kappa_0\rangle \\
\quad = \quad \mathcal{C}[\![\mathsf{jz}\ Z\ L]\!](\mathcal{C}[\![\mathsf{inc}\ X]\!](\mathcal{C}[\![\mathsf{label}\ L]\!]\langle \iota, \kappa_0\rangle)) \\
\quad = \quad \mathcal{C}[\![\mathsf{jz}\ Z\ L]\!](\mathcal{C}[\![\mathsf{inc}\ X]\!]\langle \iota[L \mapsto \kappa_0], \kappa_0\rangle) \\
\quad = \quad \mathcal{C}[\![\mathsf{jz}\ Z\ L]\!]\langle \iota[L \mapsto \kappa_0], \lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1]\rangle \\
\quad = \quad \langle \iota[L \mapsto \kappa_0], \lambda\sigma'.\ \mathsf{if}\ \sigma'(Z) = 0 \\
\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ (\iota[L \mapsto \kappa_0]\ L)\ \sigma' \\
\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma'\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \mathcal{C}[\![\mathsf{jz}\ Z\ L\ ;\ \mathsf{inc}\ X\ ;\ \mathsf{label}\ L]\!]\langle \iota, \kappa_0\rangle) \\
\quad = \quad \mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \langle \iota[L \mapsto \kappa_0], \lambda\sigma'.\ \mathsf{if}\ \sigma'(Z) = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ (\iota[L \mapsto \kappa_0]\ L)\ \sigma' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma'\rangle) \\
\quad = \quad \langle \bot_{Jump}[L \mapsto \kappa_0], \lambda\sigma'.\ \mathsf{if}\ \sigma'(Z) = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ (\bot_{Jump}[L \mapsto \kappa_0]\ L)\ \sigma' \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma'\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![\mathsf{jz}\ Z\ L\ ;\ \mathsf{inc}\ X\ ;\ \mathsf{label}\ L]\!] \\
\quad = \quad (\pi_2\ (\mathsf{fix}\ (\lambda\langle \iota, \kappa\rangle.\ \mathcal{C}[\![\mathsf{jz}\ Z\ L\ ;\ \mathsf{inc}\ X\ ;\ \mathsf{label}\ L]\!]\langle \iota, \kappa_0\rangle)))\ \sigma_0 \\
\quad = \quad (\pi_2\ \langle \bot_{Jump}[L \mapsto \kappa_0], \lambda\sigma'.\ \mathsf{if}\ \sigma'(Z) = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ (\bot_{Jump}[L \mapsto \kappa_0]\ L)\ \sigma' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma'\rangle)\ \sigma_0 \\
\quad = \quad (\lambda\sigma'.\ \mathsf{if}\ \sigma'(Z) = 0 \\
\qquad\qquad \mathsf{then}\ (\bot_{Jump}[L \mapsto \kappa_0]\ L)\ \sigma' \\
\qquad\qquad \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma')\ \sigma_0 \\
\quad = \quad \mathsf{if}\ \sigma_0(Z) = 0\ \mathsf{then}\ (\bot_{Jump}[L \mapsto \kappa_0]\ L)\ \sigma_0\ \mathsf{else}\ (\lambda\sigma.\ \kappa_0\ \sigma[X \mapsto \sigma(X)+1])\ \sigma_0 \\
\quad = \quad (\bot_{Jump}[L \mapsto \kappa_0]\ L)\ \sigma_0 \\
\quad = \quad \kappa_0\ \sigma_0 \\
\quad = \quad \lfloor \sigma_0 \rfloor
\end{array}
$$

iii. (3 pts.) $\mathcal{A}[\![\mathsf{label}\ L\ ;\ \mathsf{jz}\ Z\ L]\!]$

$$
\begin{array}{l}
\mathcal{C}[\![\mathsf{label}\ L\ ;\ \mathsf{jz}\ Z\ L]\!]\langle\iota,\kappa_0\rangle \\
\quad = \quad \mathcal{C}[\![\mathsf{label}\ L]\!](\mathcal{C}[\![\mathsf{jz}\ Z\ L]\!]\langle\iota,\kappa_0\rangle) \\
\quad = \quad \mathcal{C}[\![\mathsf{label}\ L]\!]\langle\iota,\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma\rangle \\
\quad = \quad \left\langle \begin{array}{l} \iota[L\mapsto\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma], \\ \lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma \end{array} \right\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle.\ \mathcal{C}[\![\mathsf{label}\ L\ ;\ \mathsf{jz}\ Z\ L]\!]\langle\iota,\kappa_0\rangle) \\
\quad = \quad \mathsf{fix}\ \left(\lambda\langle\iota,\kappa_0\rangle.\ \left\langle \begin{array}{l} \iota[L\mapsto\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma], \\ \lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma \end{array} \right\rangle\right) \\
\quad = \quad \langle\iota_f,\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota_f\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma\rangle \\
\qquad \text{where } \iota_f = \bot_{Jump}[L\mapsto\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ \bot_{Cont}\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![\mathsf{label}\ L\ ;\ \mathsf{jz}\ Z\ L]\!] \\
\quad = \quad (\pi_2\ (\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle.\ \mathcal{C}[\![\mathsf{label}\ L\ ;\ \mathsf{jz}\ Z\ L]\!]\langle\iota,\kappa_0\rangle)))\ \sigma_0 \\
\quad = \quad (\pi_2\ \langle\iota_f,\mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota_f\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma\rangle)\ \sigma_0 \\
\quad = \quad (\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ (\iota_f\ L)\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma)\ \sigma_0 \\
\quad = \quad \mathsf{if}\ \sigma_0(Z)=0\ \mathsf{then}\ (\iota_f\ L)\ \sigma_0\ \mathsf{else}\ \kappa_0\ \sigma_0 \\
\quad = \quad (\iota_f\ L)\ \sigma_0 \\
\quad = \quad (\bot_{Jump}[L\mapsto\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ \bot_{Cont}\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma]\ L)\ \sigma_0 \\
\quad = \quad (\lambda\sigma.\ \mathsf{if}\ \sigma(Z)=0\ \mathsf{then}\ \bot_{Cont}\ \sigma\ \mathsf{else}\ \kappa_0\ \sigma)\ \sigma_0 \\
\quad = \quad \mathsf{if}\ \sigma_0(Z)=0\ \mathsf{then}\ \bot_{Cont}\ \sigma_0\ \mathsf{else}\ \kappa_0\ \sigma_0 \\
\quad = \quad \bot_{Cont}\ \sigma_0 \\
\quad = \quad \bot_{Store_\bot}
\end{array}
$$

d. (5 pts.) Explain why the following definition of $\mathcal{A}[\![c]\!]$ does not yield the desired semantics:

$$\mathcal{A}[\![c]\!] \quad = \quad (\pi_2\ (\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle \in Jump\times Cont.\ \mathcal{C}[\![c]\!]\langle\iota,\kappa\rangle)))\ \sigma_0$$

Suppose we used the definition of $\mathcal{A}$ given above. Then,

$$\mathsf{fix}\ (\lambda\langle\iota,\kappa\rangle.\ \mathcal{C}[\![c]\!]\langle\iota,\kappa\rangle) = \langle\iota_f,\kappa_f\rangle$$

such that $\langle\iota_f,\kappa_f\rangle = \mathcal{C}[\![c]\!]\langle\iota_f,\kappa_f\rangle$. Therefore, κ_f is the continuation which accepts a store σ, evaluates c using ι_f in σ, and passes the resulting store σ' to κ_f. Hence, the semantics given by the definition of $\mathcal{A}$ above correspond to a "looping program," where the program is continually reevaluted in the store which was produced in the previous evaluation. Thus, $\kappa_f = \bot_{Cont}$ and every program has the denotation $\bot_{Store_\bot}$, clearly not the desired semantics.