# IMP and Denotational Semantics

Andrew Tolmach

June 1, 2024

To study the essence of imperative languages, many authors use a somewhat different system than TAPL's Ch. 13 STLC+References. This may go under the name IMP (for "*Imp*erative Language"), LC ("*L*anguage of *C*ommands") or just "While language" (for reasons that will become clear). Although details may vary, the basic idea of the language is that programs are made up of *commands*, which operate on a *state* that maps mutable variables to values. There is no separate notion of first-class references. In the simplest versions of IMP there are no functions, and hence perforce no recursion. On the other hand, there is a `while` command that performs (possibly infinite) iteration, so the language is Turing-complete.

In addition to giving a simple account of mutable variables that is close to the behavior of many imperative languages, IMP is also a good vehicle for introducing *denotational semantics*, a semantic framework that is much less "syntactic" than the operational semantics we have seen so far. (IMP is also widely used to introduce *axiomatic semantics* (or *programming logics*) used when proving properties of programs, but we will not cover these.)

## 1 Syntax of IMP

Commands, which do not produce a value, are syntactically distinguished from arithmetic and boolean expressions, which do.

| | | | |
|---|---|---|---|
| a | ::= | `x` | |
| | | `0` | |
| | | `succ a` | |
| | | `pred a` | |
| b | ::= | `true` | |
| | | `false` | |
| | | `not b` | |
| | | `iszero a` | |
| c | ::= | `skip` | do nothing |
| | | `x := a` | assignment |
| | | $\texttt{c}_1\texttt{; c}_2$ | sequencing |
| | | `if b then c else c` | conditional |
| | | `while b do c` | iteration |

For simplicity and familiarity, we will take the arithmetic and boolean expressions to be those of

TAPL Ch. 3, except that (i) `if-then-else` is now a command rather than a boolean expression; (ii) we add a `not` operator on booleans; and (iii) variables can only hold numbers, not booleans. (Most presentations of IMP support richer sets of arithmetic and boolean operators, but this is inessential.) Variables all have global scope (there is no mechanism for declaring them).

Our main focus is on the set of commands, which are intended to have the usual intuitive semantics. The `skip` command is useful for forming one-armed conditionals (e.g., that do nothing in the `else` case) and is also needed in intermediate forms that arise during small-step evaluation. As given, the syntax for commands is ambiguous (because of the sequencing operator); we will feel free to add parentheses where necessary to clarify the intended parse.

Note that because arithmetic and boolean expressions are syntactically disjoint, it is impossible to construct an (intuitively) ill-typed term. Again, this restriction is not essential, but it avoids the need to give explicit typing rules—although it is straightforward to do so, bearing im mind that commands do not have types (since they do not produce values).

EXAMPLE: The following (silly) program sums the initial values of `y` and `z`, leaving the result in both variables:

```
while not (iszero y) do (z := succ z; y := pred y); y := z
```

## 2 Big-step Semantics

It is not hard to give operational stepping rules for IMP, once we fix a mechanism for describing the state. We follow the scheme of TAPL Ch. 13 by tracking a state $\sigma$ along with the term being reduced. However, for IMP, the state is a mapping directly from variable names to (arithmetic) values, without any intermediate notion of "locations." (Alternatively, one can think of the variables as *being* locations.) We use $\sigma$ instead of $\mu$ to represent states, to emphasize that the domain of the mapping is slightly different.

We choose to give a big-step semantics, as these rules are slightly simpler than the small-step ones, and make a better comparison with the denotational approach to come later. We require three separate relations, one for each syntactic category. The expression evaluation relations $\Downarrow_a$ and $\Downarrow_b$ relate an expression and an initial state to a final value; evaluation never changes the state. The command evaluation relation $\Downarrow_c$ relates a command and an initial state to (just) a final state; evaluation does not produce a value.

Arithmetic (numeric values `nv` are as in TAPL Ch.3):

$$\frac{\sigma(\mathtt{x}) = \mathtt{v}}{\mathtt{x} \mid \sigma \Downarrow_a \mathtt{v}} \qquad \text{(B-VAR)}$$

$$\mathtt{0} \mid \sigma \Downarrow_a \mathtt{0} \qquad \text{(B-ZERO)}$$

$$\frac{\mathtt{a} \mid \sigma \Downarrow_a \mathtt{nv}}{\mathtt{succ\ a} \mid \sigma \Downarrow_a \mathtt{succ\ nv}} \qquad \text{(B-SUCC)}$$

$$\frac{\mathtt{a} \mid \sigma \Downarrow_a \mathtt{0}}{\mathtt{pred\ a} \mid \sigma \Downarrow_a \mathtt{0}} \qquad \text{(B-PREDZERO)}$$

$$\frac{\texttt{a} \mid \sigma \Downarrow_a \texttt{succ nv}}{\texttt{pred a} \mid \sigma \Downarrow_a \texttt{nv}} \quad \text{(B-PredSucc)}$$

Note that we treat states $\sigma$ as being finite; a program that tries to read the value of an undefined variable will have no semantics at all (since the premise of B-Var will fail).

Booleans:

$$\texttt{true} \mid \sigma \Downarrow_b \texttt{true} \quad \text{(B-True)}$$

$$\texttt{false} \mid \sigma \Downarrow_b \texttt{false} \quad \text{(B-False)}$$

$$\frac{\texttt{b} \mid \sigma \Downarrow_b \texttt{true}}{\texttt{not b} \mid \sigma \Downarrow_b \texttt{false}} \quad \text{(B-NotTrue)}$$

$$\frac{\texttt{b} \mid \sigma \Downarrow_b \texttt{false}}{\texttt{not b} \mid \sigma \Downarrow_b \texttt{true}} \quad \text{(B-NotFalse)}$$

$$\frac{\texttt{a} \mid \sigma \Downarrow_a \texttt{0}}{\texttt{iszero a} \mid \sigma \Downarrow_b \texttt{true}} \quad \text{(B-IszeroZero)}$$

$$\frac{\texttt{a} \mid \sigma \Downarrow_a \texttt{succ nv}}{\texttt{iszero a} \mid \sigma \Downarrow_b \texttt{false}} \quad \text{(B-IszeroSucc)}$$

Commands:

$$\texttt{skip} \mid \sigma \Downarrow_c \sigma \quad \text{(B-Skip)}$$

$$\frac{\texttt{a} \mid \sigma \Downarrow_a \texttt{nv}}{\texttt{x := a} \mid \sigma \Downarrow_c [\texttt{x} \mapsto \texttt{nv}]\sigma} \quad \text{(B-Assign)}$$

$$\frac{\texttt{c}_1 \mid \sigma \Downarrow_c \sigma_1 \qquad \texttt{c}_2 \mid \sigma_1 \Downarrow_c \sigma_2}{\texttt{c}_1\texttt{; c}_2 \mid \sigma \Downarrow_c \sigma_2} \quad \text{(B-Seq)}$$

$$\frac{\texttt{b}_1 \mid \sigma \Downarrow_b \texttt{true} \qquad \texttt{c}_2 \mid \sigma \Downarrow_c \sigma_2}{\texttt{if b}_1 \texttt{ then c}_2 \texttt{ else c}_3 \mid \sigma \Downarrow_c \sigma_2} \quad \text{(B-IfTrue)}$$

$$\frac{\texttt{b}_1 \mid \sigma \Downarrow_b \texttt{false} \qquad \texttt{c}_3 \mid \sigma \Downarrow_c \sigma_3}{\texttt{if b}_1 \texttt{ then c}_2 \texttt{ else c}_3 \mid \sigma \Downarrow_c \sigma_3} \quad \text{(B-IfFalse)}$$

$$\frac{\texttt{b}_1 \mid \sigma \Downarrow_b \texttt{false}}{\texttt{while b}_1 \texttt{ do c}_2 \mid \sigma \Downarrow_c \sigma} \quad \text{(B-WhileFalse)}$$

$$\frac{\texttt{b}_1 \mid \sigma \Downarrow_b \texttt{true} \qquad \texttt{c}_2 \mid \sigma \Downarrow_c \sigma_2 \qquad \texttt{while b}_1 \texttt{ do c}_2 \mid \sigma_2 \Downarrow_c \sigma'}{\texttt{while b}_1 \texttt{ do c}_2 \mid \sigma \Downarrow_c \sigma'} \quad \text{(B-WhileTrue)}$$

The most interesting case is for `while`. Its semantics are inspired by the observation that

$$\texttt{while } b_1 \texttt{ do } c_2$$

should behave the same way as its "one step unfolding"

$$\texttt{if } b_1 \texttt{ then } (c_2\texttt{; while } b_1 \texttt{ do } c_2) \texttt{ else skip}.$$

At first sight, the B-WhileTrue rule might seem circular in its third premise: how can we evaluate a term by appealing recursively to the evaluation of the same term? The answer is that the premise does the evaluation in the (potentially) *different* state $\sigma_2$ that results from evaluating the body of the `while`. In particular, if evaluating the body changes the state so as to make the test condition $b_1$ become false, the recursive invocation will use rule B-WhileFalse and the loop will terminate. (On the other hand, if the condition never does go false, we will never be able to construct a finite derivation, so this semantics won't tell us anything about the behavior of the `while` command; this is the same problem we encountered with big-step semantics for the $\lambda$-calculus.) So B-WhileTrue is a perfectly reasonable rule. However, it does have one drawback. Every other rule for IMP is *compositional*, which means that its premises only mention properties of *sub-terms* of term in its conclusion. Compositional rules are nice because they support reasoning by structural induction over terms. B-WhileTrue is *not* compositional, since the entire `while` term from the conclusion appears unchanged in the third premise. (The application rule for the $\lambda$-calculus is similarly non-compositional.)

Exercise 1: Give a *small-step* semantics for IMP. The stepping rules for arithmetic and boolean expressions are completely straightforward with the addition of a memory as in the big-step case. To evaluate commands, it is useful to treat `skip` as a normal form, to which other commands (notably assignment) may reduce; it plays a role analogous to `unit` in STLC. To evaluate `while`, use the one-step unfolding trick above.

## 3 A Taste of Denotational Semantics

Denotational semantics is an approach is which we give each term in the language a meaning (its "*denotation*") in a mathematically-defined *semantic domain*. Unlike the rewriting-based operational semantics we have seen so far, which are basically all about manipulating syntax, denotational semantics lets us describe program meaning directly in terms of mathematical abstractions (often functions). This can make it easier to compare the meanings of different programs, in particular if they are written in different languages.

A denotational semantics is defined by giving a *meaning function* for each different kind of term, mapping it to a suitable semantic domain. For example, if we consider just a language of arithmetic expressions with no variables, the semantic domain might be the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, and the meaning function would assign a value $n \in \mathbb{N}$ to every expression. Typically this will involve recursive invocation of the meaning function on sub-expressions. We require all meaning functions to be compositional; among other things, this guarantees that any recursions are well-founded.

For IMP, the value of an arithmetic or boolean expression depends on the values of variables in the state, so the meaning of these expressions will themselves be *functions* from states to naturals ($\mathbb{N}$) and booleans ($\mathbb{B} = \{\mathit{True}, \mathit{False}\}$) respectively. (Here we write *True* and *False* in italics to emphasize that these are truth values in an abstract mathematical set, rather than terms in the original language.) Similarly, a state $\sigma \in \Sigma$ itself is now a function from the set of variable names

(*Var*) to natural numbers ($\mathbb{N}$) (rather than to terms `nv`). Also, for simplicity, we now treat states as *total* functions defined on all *possible* variable names; for example, we might choose to define an initial environment mapping all variables to the natural number 0.

Finally, what should be the meaning of an IMP command? We have already observed that the effect of evaluating a command is simply to modify the state. This suggests defining the meaning as being a function from states to states ($\Sigma \to \Sigma$). (A function like this that itself takes a function as an argument and returns a function as a result is sometimes called a *functional*, or, more generally, a "higher-order" function.) But recalling that IMP contains a `while` construct, we can foresee a need to define the meaning of non-terminating (or *diverging*) computations, so we actually use a slightly richer notion. We will write $\perp$ (pronounced "bottom" or "diverge") to represent an "undefined" or "unknown" state, and write $\Sigma_\perp$ for $\Sigma \cup \{\perp\}$, i.e. the extension of the state domain with a bottom element. (This is sometimes called the "lifted" state domain. More generally, we can define $S_\perp = S \cup \{\perp\}$ for any set $S$.) The meaning of IMP commands will be be functions of the form $\Sigma \to \Sigma_\perp$. (Alternatively, these can be viewed as *partial* functions from states to states, written $\Sigma \rightharpoonup \Sigma$, where a function is considered to be undefined on values that map to $\perp$.)

It is traditional to use double square brackets ($[\![]\!]$), sometimes called "Oxford brackets," to specify the meaning function for each term. For example, the meaning function for IMP arithmetic expressions is:

$$
\begin{array}{lcl}
[\![\texttt{a}]\!]_a & : & \Sigma \to \mathbb{N} \\
[\![\texttt{x}]\!]_a & = & \lambda\sigma.\sigma(\texttt{x}) \\
[\![\texttt{0}]\!]_a & = & \lambda\sigma.0 \\
[\![\texttt{succ a}]\!]_a & = & \lambda\sigma.[\![\texttt{a}]\!]_a(\sigma) + 1 \\
[\![\texttt{pred a}]\!]_a & = & \lambda\sigma.[\![\texttt{a}]\!]_a(\sigma) \dot{-} 1
\end{array}
$$

Note that here we are using $\lambda$ notation on the right-hand sides just as a convenient mechanism for writing down (anonymous) function definitions, but these are ordinary mathematical functions, *not* expressions in the lambda calculus! Similarly, the symbols 0 is the natural number zero and $+$ and $\dot{-}$ (pronounced "monus") are ordinary mathematical operations on the natural numbers.[1]

The meaning function for IMP boolean expressions is similar:

$$
\begin{array}{lcl}
[\![\texttt{b}]\!]_a & : & \Sigma \to \mathbb{B} \\
[\![\texttt{true}]\!]_a & = & \lambda\sigma.\mathit{True} \\
[\![\texttt{false}]\!]_a & = & \lambda\sigma.\mathit{False} \\
[\![\texttt{not b}]\!]_a & = & \lambda\sigma.\neg[\![\texttt{b}]\!]_b(\sigma) \\
[\![\texttt{iszero a}]\!]_a & = & \lambda\sigma.\begin{cases} \mathit{True} & \text{if } [\![\texttt{a}]\!]_a(\sigma) = 0 \\ \mathit{False} & \text{if } [\![\texttt{a}]\!]_a(\sigma) > 0 \end{cases}
\end{array}
$$

Finally, we turn to the meaning function for commands.

[1] Monus is defined as $x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$

$$
\begin{array}{lcl}
[\![\texttt{c}]\!]_c & : & \Sigma \to \Sigma_\perp \\
[\![\texttt{skip}]\!]_c & = & \lambda\sigma.\sigma \\
[\![\texttt{x := a}]\!]_c & = & \lambda\sigma.[\texttt{x} \mapsto [\![\texttt{a}]\!]_a(\sigma)]\sigma \\
[\![\texttt{c}_1\texttt{; c}_2]\!]_c & = & [\![\texttt{c}_2]\!]_c \circ_\perp [\![\texttt{c}_1]\!]_c \\
[\![\texttt{if b}_1\texttt{ then c}_2\texttt{ else c}_3]\!]_c & = & \lambda\sigma.\begin{cases} [\![\texttt{c}_2]\!]_c(\sigma) & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{True} \\ [\![\texttt{c}_3]\!]_c(\sigma) & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{False} \end{cases} \\
[\![\texttt{while b}_1\texttt{ do c}_2]\!]_c & = & \mathit{fix}(F) \\
 & & \text{where } F = \lambda w.\lambda\sigma.\begin{cases} \sigma & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{False} \\ (w \circ_\perp [\![\texttt{c}_2]\!]_c)(\sigma) & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{True} \end{cases}
\end{array}
$$

Let us consider each command in turn.

- Since `skip` does nothing, its denotation is just the identity function on states.
- `x := a` updates the state by changing (just) the value corresponding to `x`. The value of `a` is computed in the original (unchanged) state.
- The sequence of commands $\texttt{c}_1\texttt{; c}_2$ first modifies the state according to $\texttt{c}_1$ and then according to $\texttt{c}_2$. Formally, it is written as a slightly non-standard form of function composition that propagates $\perp$:
$$f \circ_\perp g = \lambda x.\begin{cases} \perp & \text{if } g(x) = \perp \\ f(g(x)) & \text{otherwise} \end{cases}$$
- The effect of a conditional depends on the truth value of its test expression, in the obvious way.
- The semantics of a `while` loop are again based on the notion of a one-step unfolding. We would like the semantics defined so that
$$[\![\texttt{while b}_1\texttt{ do c}_2]\!]_c = [\![\texttt{if b}_1\texttt{ then (c}_2\texttt{; while b}_1\texttt{ do c}_2\texttt{) else skip}]\!]_c$$
Setting $w = [\![\texttt{while b}_1\texttt{ do c}_2]\!]_c$ and unfolding the denotation function on the right-hand side, we get
$$w = \lambda\sigma.\begin{cases} \sigma & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{False} \\ (w \circ_\perp [\![\texttt{c}_2]\!]_c)(\sigma) & \text{if } [\![\texttt{b}_1]\!]_b(\sigma) = \mathit{True} \end{cases}$$
To solve this recursive equation, we abstract over $w$ to define the functional $F : (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$, as shown above, and then use the *fix* operator to find the least fixed point of $F$. Since $F$ is now a genuine mathematical function, we need to confirm that it actually *has* a fixed point. (Not every function does; e.g. consider $\lambda x.(x+1)$.) As we'll discuss in more detail later, $F$ *does* have a (least) fixed point, but it might turn out to be the function $\lambda\sigma.\perp$ that maps every initial state to the undefined state (which is why we extended the codomain of denotation functions to include $\perp$ in the first place). In particular, this will be the denotation we get for infinite loops such as `while true do skip`.

  In working with the meaning of `while` commmands, we will make heavy use of the fundamental property of *fix*, namely that $F(\mathit{fix}(F)) = \mathit{fix}(F)$.

Note that, unlike the big-step semantics, the denotational semantics for commands *is* compositional. Again, this is one of the hallmarks of denotational semantics.

EXAMPLE 1. To illustrate how the meaning function is applied and what it can be useful for, we will show that $[\![\texttt{x := succ x; x := pred x}]\!]_c = [\![\texttt{skip}]\!]_c$. First, for any $\sigma$, we have

$$\begin{aligned}[\![\texttt{x := succ x}]\!]_c(\sigma) &= [\texttt{x} \mapsto [\![\texttt{succ x}]\!]_c(\sigma)]\sigma \\ &= [\texttt{x} \mapsto [\![\texttt{x}]\!]_c(\sigma) + 1]\sigma \\ &= [\texttt{x} \mapsto \sigma(\texttt{x}) + 1]\sigma \end{aligned} \tag{1}$$

Similarly,

$$[\![\texttt{x := pred x}]\!]_c(\sigma) = [\texttt{x} \mapsto \sigma(\texttt{x}) \dot{-} 1]\sigma \tag{2}$$

We also note (without proof) several useful generic properties of state update:

$$([\texttt{x} \mapsto n]\sigma)(\texttt{x}) = n \tag{3}$$

$$[\texttt{x} \mapsto n_2]([\texttt{x} \mapsto n_1]\sigma) = [\texttt{x} \mapsto n_2]\sigma \tag{4}$$

$$[\texttt{x} \mapsto \sigma(\texttt{x})]\sigma = \sigma \tag{5}$$

Then, for any $\sigma$, we can compute:

| | | |
|---|---|---|
| | $[\![\texttt{x := succ x; x := pred x}]\!]_c(\sigma)$ | |
| = | $([\![\texttt{x := pred x}]\!]_c \circ_\perp [\![\texttt{x := succ x}]\!]_c)(\sigma)$ | by definition of $[\![]\!]_c$ |
| = | $[\![\texttt{x := pred x}]\!]_c(\sigma')$ | by (1) and definition of $\circ_\perp$, where $\sigma' = [\texttt{x} \mapsto \sigma(\texttt{x}) + 1]\sigma$ |
| = | $[\texttt{x} \mapsto \sigma'(\texttt{x}) \dot{-} 1]\sigma'$ | by (2) |
| = | $[\texttt{x} \mapsto (\sigma(\texttt{x}) + 1) \dot{-} 1]\sigma'$ | by definition of $\sigma'$ and (3) |
| = | $[\texttt{x} \mapsto \sigma(\texttt{x})]\sigma'$ | by arithmetic |
| = | $[\texttt{x} \mapsto \sigma(\texttt{x})]\sigma$ | by definition of $\sigma'$ and (4) |
| = | $\sigma$ | by (5) |
| = | $[\![\texttt{skip}]\!]_c(\sigma)$ | by definition of $[\![]\!]_c$ |

□

EXERCISE 2. Prove the following identity:

$$[\![\texttt{x := 0; if iszero y then x := y else y := x}]\!]_c = [\![\texttt{x := 0; y := 0}]\!]_c$$

EXAMPLE 2. Let $\sigma_0$ be the state that maps every variable to 0. To show how to compute using fixed points, we prove that if $\sigma_1 = [\texttt{x} \mapsto 2]\sigma_0$ then $[\![\texttt{while not (iszero x) do x := pred x}]\!]_c(\sigma_1) = \sigma_0$.

To start with, we have

$$[\![\texttt{while not (iszero x) do x := pred x}]\!]_c = \mathit{fix}(F)$$

where

$$F = \lambda w.\lambda\sigma.\mathit{if}\ [\![\texttt{not (iszero x)}]\!]_b(\sigma)\ \mathit{then}\ (w \circ_\perp [\![\texttt{x := pred x}]\!]_c)(\sigma)\ \mathit{else}\ \sigma$$

For any state $\sigma$ we have

$$\begin{aligned}[\![\texttt{not (iszero x)}]\!]_b(\sigma) &= \neg[\![\texttt{iszero x}]\!]_b(\sigma) \\ &= \neg([\![\texttt{x}]\!]_a(\sigma) = 0) \\ &= [\![\texttt{x}]\!]_a(\sigma) > 0 \end{aligned} \tag{6}$$

Using this and (2) from the previous example, we have

$$F = \lambda w.\lambda\sigma.\mathit{if}\ \sigma(\mathtt{x}) > 0\ \mathit{then}\ w([\mathtt{x} \mapsto \sigma(\mathtt{x}) \dot{-} 1]\sigma)\ \mathit{else}\ \sigma \tag{7}$$

Now, in particular, we can compute

$$\begin{array}{lll}
& [\![\texttt{while not (iszero x) do x := pred x}]\!]_c(\sigma_1) & \\
= & \mathit{fix}(F)(\sigma_1) & \\
= & F(\mathit{fix}(F))(\sigma_1) & \text{by fundamental property of } \mathit{fix} \\
= & \mathit{if}\ \sigma_1(\mathtt{x}) > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto \sigma_1(\mathtt{x}) \dot{-} 1]\sigma_1)\ \mathit{else}\ \sigma_1 & \text{by (7)} \\
= & \mathit{if}\ 2 > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto 2 \dot{-} 1]\sigma_1)\ \mathit{else}\ \sigma_1 & \text{by definition of } \sigma_1 \\
= & \mathit{fix}(F)(\sigma_2) & \text{where } \sigma_2 = [\mathtt{x} \mapsto 1]\sigma_1 \\
= & F(\mathit{fix}(F))(\sigma_2) & \text{by fundamental property of } \mathit{fix} \\
= & \mathit{if}\ \sigma_2(\mathtt{x}) > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto \sigma_2(\mathtt{x}) \dot{-} 1]\sigma_2)\ \mathit{else}\ \sigma_2 & \text{by (7)} \\
= & \mathit{if}\ 1 > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto 1 \dot{-} 1]\sigma_2)\ \mathit{else}\ \sigma_2 & \text{by definition of } \sigma_2 \\
= & \mathit{fix}(F)(\sigma_3) & \text{where } \sigma_3 = [\mathtt{x} \mapsto 0]\sigma_2 \\
= & F(\mathit{fix}(F))(\sigma_3) & \text{by fundamental property of } \mathit{fix} \\
= & \mathit{if}\ \sigma_3(\mathtt{x}) > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto \sigma_3(\mathtt{x}) \dot{-} 1]\sigma_3)\ \mathit{else}\ \sigma_3 & \text{by (7)} \\
= & \mathit{if}\ 0 > 0\ \mathit{then}\ \mathit{fix}(F)([\mathtt{x} \mapsto 0 \dot{-} 1]\sigma_3)\ \mathit{else}\ \sigma_3 & \text{by definition of } \sigma_3 \\
= & \sigma_3 & \\
= & [x \mapsto 0]([x \mapsto 1]([x \mapsto 2]\sigma_0)) & \text{by definitions of } \sigma_3, \sigma_2, \sigma_1 \\
= & \sigma_0 & \text{by (4) and (5)}
\end{array}$$

□

# 4 Least Fixed Point Semantics

We have used the *fix* operator to define the denotational semantics of `while` statements, and done calculations relying on the fundamental property of fixed points, $F(\mathit{fix}(F)) = \mathit{fix}(F)$. But we have not not yet established that the relevant fixed point actually exists, nor have we explored how undefined states ($\perp$) arise. We now address these issues.

The basic idea is that we can solve a recursive equation for a function by considering *partial unfoldings* of the function, and taking their union. The actual recursive function will be the limit we reach as the number of unfoldings goes to infinity. This limit is the least fixed point of the functional corresponding to our recursive function.

Recall that the recursive equation we wish to solve has the form:

$$w(\sigma) = \mathit{if}\ b(\sigma)\ \mathit{then}\ w(c(\sigma))\ \mathit{else}\ \sigma$$

where $w = [\![\texttt{while b}_1\ \texttt{do c}_2]\!]_c$, $b = [\![\texttt{b}_1]\!]_b$ and $c = [\![\texttt{c}_2]\!]_c$, and we are using "*if-then-else*" notation as shorthand for the two cases in the mathematical definition of $[\![]\!]_c$ on `if` commands. Our goal is to show that a $w$ satisfying this equation exists, and to give a way to compute it.

To attack this goal, it is very helpful to take a slightly alternative viewpoint on functions. Any (total) function $F : A \to B$ can be viewed as a binary relation $R \subseteq A \times B$, where $(a, b) \in R$ iff $F(a) = b$. A partial function $F : A \rightharpoonup B$ can be viewed as a relation $R$ in the same way, where $R$ contains a pair for the form $(a, b)$ only if $F$ is defined on $a$. $R$ is called the *graph* of $F$, which

we write $R = \lceil F \rceil$. Note that $R$ is just a set of ordered pairs, with the property that no two pairs in the set have the same first element (i.e., if $(a, b_1) \in R$ and $(a, b_2) \in R$, then $b_1 = b_2$). In many cases of interest, $R$ will be an infinite set. It should be clear that two functions are equal (in the sense of taking equal arguments to equal results) iff their graphs are equal (as sets); this is called the *extensionality principle*.

Specializing to our situation here, recall that $w$ is a total function from states $\Sigma$ to "lifted states" $\Sigma_\perp$. Intuitively, $\perp$ represents a state that is undefined, or, better, about which we have "no information." We can also view $w$ as a partial function from $\Sigma$ to $\Sigma$, which is considered to be undefined on $\sigma$ iff $w(\sigma) = \perp$. We'll find it convenient to write $w$ as a total function, but to take its graph to be that of the partial function, i.e. the graph is a relation on $\Sigma \times \Sigma$ that contains $(\sigma, w(\sigma'))$ pairs only for those $\sigma$ that $w$ does not map to $\perp$.

Since $\Sigma$ is infinite (there are an infinite number of variable names, each of which might hold any natural number), it should be clear that the graph of $w$ is also infinite. It is often useful to build a description of an infinite set in stages, by considering finite subsets. In the case of function $w$, we will consider the subsets defined by allowing a fixed, finite number of unfoldings of the recursive definition. Each of these allows us to define the behavior of $w$ on a larger group of initial states.

Let's start by considering a version of $w$ that allows exactly one unfolding, i.e. we can see the right-hand side, but we cannot compute a recursive application. If we try to compute such a call, we arrange to get $\perp$, representing "no information." We can write this as as a non-recursive function:

$$
\begin{array}{ll}
w_1(\sigma) = & \mathit{if}\ b(\sigma)\ \mathit{then} \\
& \quad (\lambda\_.\perp)(c(\sigma)) \\
& \mathit{else} \\
& \quad \sigma
\end{array}
$$

Note the use of $\lambda\_.\perp$ where the recursive call to $w$ used to be: essentially, the result of $c(\sigma)$ is ignored, and this function simply returns "don't know." Think of $w_1$ as a (not very good) *approximation* of the full definition of $w$. It doesn't say much, but it *does* allow us to compute the behavior of $w$ on initial states $\sigma$ for which $b(\sigma)$ is false. Indeed, the graph $\lceil w_1 \rceil$ is the set $\{(\sigma, \sigma) \mid \neg b(\sigma)\}$, where we use *set comprehension* notation which can be read as "the set of pairs of the form $(\sigma, \sigma)$ such that $\neg b(\sigma)$ holds (i.e. for which $b(\sigma) = \mathit{false}$)."

Now consider what happens if we allow exactly *two* unfoldings. We get:

$$
\begin{array}{ll}
w_2(\sigma) = & \mathit{if}\ b(\sigma)\ \mathit{then} \\
& \quad \mathit{if} b(c(\sigma))\ \mathit{then} \\
& \quad\quad (\lambda\_.\perp)(c(c(\sigma))) \\
& \quad \mathit{else} \\
& \quad\quad c(\sigma) \\
& \mathit{else} \\
& \quad \sigma
\end{array}
$$

with graph $\lceil w_2 \rceil = \{(\sigma, \sigma) \mid \neg b(\sigma)\} \cup \{(\sigma, c(\sigma)) \mid \neg b(c(\sigma)))\}$. Note that $\lceil w_1 \rceil \subseteq \lceil w_2 \rceil$, i.e., $w_2$ tells us more about the behavior of $w$, while remaining consistent with what $w_1$ already told us.

Obviously, this pattern can be repeated to define $w_i$ for any finite $i$. But these definitions, while they remain finite and non-recursive, quickly grow unpleasantly large. Looking more closely at $w_2$, we see that it can be rewritten more concisely in terms of $w_1$:

$$
\begin{array}{lll}
w_2(\sigma) = & \textit{if } b(\sigma) \textit{ then} \\
& \quad w_1(c(\sigma)) \\
& \textit{else} \\
& \quad \sigma
\end{array}
$$

By defining a suitable version of $w$ corresponding to *zero* unfoldings (i.e. we cannot see the right hand side at all) we can then write a general scheme defining $w_i$ for all natural numbers $i$.

$$
\begin{array}{lll}
w_0(\sigma) & = & \bot \\
w_{i+1}(\sigma) & = & \textit{if } b(\sigma) \textit{ then} \\
& & \quad w_i(c(\sigma)) \\
& & \textit{else} \\
& & \quad \sigma
\end{array}
$$

In terms of graphs, we have:

$$
\begin{array}{lll}
\lceil w_0 \rceil & = & \emptyset \\
\lceil w_{i+1} \rceil & = & \lceil w_i \rceil \cup \{(\sigma, c^i(\sigma)) \mid \neg b(c^i(\sigma)\}
\end{array}
$$

where $c^i$ is $i$-fold application of $c$, i.e. $c^i = c \circ c \circ \cdots \circ c$, $i$ times.

Now what we would like to to do is to define $w$ as the function that combines the behavior of all these partial definitions, i.e. a kind of limit as $i$ approaches infinity. Mathematically, it is easiest to express this in terms of the functions' graphs, using the idea of an infinite union of sets. Take

$$W = \bigcup_{i=0}^{\infty} \lceil w_i \rceil$$

Since each of the graphs in the union represents a partial function on states, it should be clear that $W$ does too; i.e. $W = \lceil w \rceil$ for some function $w : \Sigma \to \Sigma_\bot$. This $w$ is the desired solution to our recursive equation. Using it will let us compute the state that results from executing any `while` loop that iterates an arbitrary, finite number of times before $b$ turns false. For an infinite loop, $w = w_0$, which has the empty graph, i.e. it will give $\bot$ as the result state for any initial state.

What does this all this have to do with fixed points? Well, another way to write the partial versions of $w$ is to extract the (non-recursive) functional

$$F = \lambda u.\lambda\sigma.\textit{if } b(\sigma) \textit{ then } u(c(\sigma)) \textit{ else } \sigma$$

and then write

$$
\begin{array}{lll}
w_0 & = & \lambda\sigma.\bot \\
w_{i+1} & = & F(w_i) \\
& = & F^i(w_0)
\end{array}
$$

where $F^i = F \circ F \circ \cdots \circ F$, $i$ times.

Thus we can write

$$\lceil w \rceil = W = \bigcup_{i=0}^{\infty} \lceil F(w_i) \rceil = \bigcup_{i=0}^{\infty} \lceil F^i(w_0) \rceil$$

Then it turns out that $w = \mathit{fix}(F)$, the *least fixed point* of $F$, where fixed point means that $F(w) = w$ and "least" means "being defined on the fewest states," or, in graph terms, as "being the smallest

set." We now proceed to prove this. As before, it is easiest to work with the graphs explicitly, and show that $\lceil F(w) \rceil = \lceil w \rceil$. We can state the effect of $F$ on graphs of arbitrary functions $u$ as

$$\lceil F(u) \rceil = \{(\sigma, \sigma') \mid b(\sigma) \wedge (c(\sigma), \sigma') \in \lceil u \rceil\} \cup \{(\sigma, \sigma) \mid \neg b(\sigma)\} \tag{8}$$

THEOREM. $w = \mathit{fix}(F)$, the least fixed point of $F$, i.e.

- $\lceil w \rceil \subseteq \lceil F(w) \rceil$;
- $\lceil F(w) \rceil \subseteq \lceil w \rceil$; and
- for any $u$ such that $\lceil F(u) \rceil = \lceil u \rceil$, $\lceil w \rceil \subseteq \lceil u \rceil$.

Before proving the theorem, we first give a preliminary definition and lemma.

DEFINITION. A function $g$ is *monotonic* if, for any $u$, $\lceil u \rceil \subseteq \lceil u' \rceil \Rightarrow \lceil g(u) \rceil \subseteq \lceil g(u') \rceil$.

LEMMA. $F$ is monotonic.

PROOF Assume $\lceil u \rceil \subseteq \lceil u' \rceil$ and suppose $(\sigma, \sigma') \in \lceil F(u) \rceil$. Then by (8), there are two cases. Case (i): $b(\sigma)$ holds and $(c(\sigma), \sigma') \in \lceil u \rceil$. Then by the assumption, $(c(\sigma), \sigma') \in \lceil u' \rceil$, and by (8), $(\sigma, \sigma') \in \lceil F(u') \rceil$. Case (ii): $\sigma = \sigma'$ and $\neg b(\sigma)$ holds. Then by (8), $(\sigma, \sigma') \in \lceil F(u') \rceil$. □

PROOF OF THEOREM

- $\lceil w \rceil \subseteq \lceil F(w) \rceil$

  For every $i$, it is immediate that $\lceil w_i \rceil \subseteq \bigcup_{i=0}^{\infty} \lceil w_i \rceil = \lceil w \rceil$, and so, by monotonicity, $\lceil F(w_i) \rceil \subseteq \lceil F(w) \rceil$. Unioning over all $i$, we get $\lceil w \rceil = \bigcup_{i=0}^{\infty} \lceil F(w_i) \rceil \subseteq \lceil F(w) \rceil$ as required.

- $\lceil F(w) \rceil \subseteq \lceil w \rceil$

  Consider any $(\sigma, \sigma') \in \lceil F(w) \rceil$. By (8), there are two cases. Case (i): $b(\sigma)$ holds and $(c(\sigma), \sigma') \in \lceil w \rceil = \bigcup_{i=0}^{\infty} \lceil w_i \rceil$. So there must be some (finite) $n$ such that $(c(\sigma), \sigma') \in \lceil w_n \rceil$. So by (8) $(\sigma, \sigma') \in \lceil F(w_n) \rceil \subseteq \bigcup_{i=0}^{\infty} \lceil F(w_i) \rceil = \lceil w \rceil$. Case (ii): $\sigma = \sigma'$ and $\neg b(\sigma)$ holds. Then $(\sigma, \sigma') \in \lceil w_1 \rceil = \lceil F(w_0) \rceil \subseteq \bigcup_{i=0}^{\infty} \lceil F(w_i) \rceil = \lceil w \rceil$.

- For any $u$, $\lceil F(u) \rceil = \lceil u \rceil \Rightarrow \lceil w \rceil \subseteq \lceil u \rceil$.

  Let $u$ be any function such that $\lceil F(u) \rceil = \lceil u \rceil$. Then certainly $\lceil w_0 \rceil = \emptyset \subseteq \lceil u \rceil$. By monotonicity, $\lceil F(w_0) \rceil \subseteq \lceil F(u) \rceil = \lceil u \rceil$. By induction, using monotonicity at each stage, $\lceil F^i(w_0) \rceil \subseteq \lceil F(u) \rceil = \lceil u \rceil$ for every $i$. Unioning over all $i$, we get $\lceil w \rceil = \bigcup_{i=0}^{\infty} \lceil F^i(w_0) \rceil \subseteq \lceil u \rceil$, as required. □

With this proof in hand, we can now be confident that our semantics for IMP is well-defined, and that the use of the fundemantal property of *fix* illustrated in, e.g. Example 2, is legitimate.

All this seems like quite a lot of work! So an obvious question is, do we have to do a similar proof every time we want to use a recursive equation to denote the meaning of some language feature? Fortunately, the answer is no: the result we got here can be obtained directly by instantiating a much more general theory of fixed points. The classic approach works in the general setting of *pointed complete partial orders*. Its theorems tell us that $\mathit{fix}(F) = \bigcup_{i=0}^{\infty} F^i(\emptyset)$ for a large class of functions $F$. The only requirement is that $F$ is *continuous*, a slightly stronger property than being monotone (as defined above), which supports a proof along the lines of the second case of

our theorem above. In practice, almost any meaning function useful in denotational semantics will be continuous, so there is no need to develop new theory to justify taking fixed points.

Exercise 3. Extend the IMP language with a command `repeat` $c$ `until` $b$ and specify its denotational semantics in the same style as for `while`, i.e., as the fixpoint of a suitable functional. The intent is that $[\![\texttt{repeat } c \texttt{ until } b]\!]_c = [\![c\texttt{; while not } b \texttt{ do } c]\!]_c$. You do not have to prove this equivalence, but you do need to prove that the fixpoint of your functional exists. Hint: just follow the outline of the theorem proof above; only places that appeal to equation (8) need to be re-argued.

### Notes

For a (relatively) gentle introduction to the mathematical details of fixed point theory, see Schmidt [1986, Ch. 6] or Winskel [1993].

David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986. URL `https://people.cs.ksu.edu/~schmidt/text/ds0122.pdf`.

Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

## 5 Suggested Solutions To Exercises

1. Small-step semantics for IMP.

Arithmetic (numeric values `nv` are as in TAPL Ch.3):

$$\frac{\sigma(\mathtt{x}) = \mathtt{v}}{\mathtt{x} \mid \sigma \to_a \mathtt{v}} \qquad \text{(E-Var)}$$

$$\frac{\mathtt{a} \mid \sigma \to_a \mathtt{a}'}{\mathtt{succ\ a} \mid \sigma \to_a \mathtt{succ\ a}'} \qquad \text{(E-Succ)}$$

$$\mathtt{pred\ 0} \mid \sigma \to_a \mathtt{0} \qquad \text{(E-PredZero)}$$

$$\mathtt{pred\ (succ\ nv)} \mid \sigma \to_a \mathtt{nv} \qquad \text{(E-PredSucc)}$$

$$\frac{\mathtt{a} \mid \sigma \to_a \mathtt{a}'}{\mathtt{pred\ a} \mid \sigma \to_a \mathtt{pred\ a}'} \qquad \text{(E-Pred)}$$

Booleans:

$$\mathtt{not\ true} \mid \sigma \to_b \mathtt{false} \qquad \text{(E-NotTrue)}$$

$$\mathtt{not\ false} \mid \sigma \to_b \mathtt{true} \qquad \text{(E-NotFalse)}$$

$$\frac{\mathtt{b} \mid \sigma \to_b \mathtt{b}'}{\mathtt{not\ b} \mid \sigma \to_b \mathtt{not\ b}'} \qquad \text{(E-Not)}$$

$$\texttt{iszero 0} \mid \sigma \to_b \texttt{true} \qquad \text{(E-IszeroZero)}$$

$$\texttt{iszero (succ nv)} \mid \sigma \to_b \texttt{false} \qquad \text{(E-IszeroSucc)}$$

$$\frac{\texttt{a} \mid \sigma \to_a \texttt{a}'}{\texttt{iszero a} \mid \sigma \to_b \texttt{iszero a}'} \qquad \text{(E-Iszero)}$$

Commands:

Note that `skip` is a normal form; if we reach it by a sequence of steps, this indicates normal successful evaluation (analogous to reaching a value in small-step semantics of expressions).

$$\frac{\texttt{a} \mid \sigma \to_a \texttt{a}'}{\texttt{x := a} \mid \sigma \to_c \texttt{x := a}' \mid \sigma} \qquad \text{(E-Assign)}$$

$$\texttt{x := nv} \mid \sigma \to_c \texttt{skip} \mid [\texttt{x} \mapsto \texttt{nv}]\sigma \qquad \text{(E-AssignVal)}$$

$$\frac{\texttt{c}_1 \mid \sigma \to_c \texttt{c}_1' \mid \sigma'}{\texttt{c}_1\texttt{; c}_2 \mid \sigma \to_c \texttt{c}_1'\texttt{; c}_2 \mid \sigma'} \qquad \text{(E-Seq1)}$$

$$\texttt{skip; c}_2 \mid \sigma \to_c \texttt{c}_2 \mid \sigma \qquad \text{(E-SeqSkip)}$$

$$\frac{\texttt{b}_1 \mid \sigma \to_b \texttt{b}_1'}{\texttt{if b}_1\texttt{ then c}_2\texttt{ else c}_3 \mid \sigma \to_c \texttt{if b}_1'\texttt{ then c}_2\texttt{ else c}_3 \mid \sigma} \qquad \text{(E-If)}$$

$$\texttt{if true then c}_2\texttt{ else c}_3 \mid \sigma \to_c \texttt{c}_2 \mid \sigma \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then c}_2\texttt{ else c}_3 \mid \sigma \to_c \texttt{c}_3 \mid \sigma \qquad \text{(E-IfFalse)}$$

$$\texttt{while b}_1\texttt{ do c}_2 \mid \sigma \to_c \texttt{if b}_1\texttt{ then (c}_2\texttt{; while b}_1\texttt{ do c}_2\texttt{) else skip} \mid \sigma \qquad \text{(E-While)}$$

2. We want to show that $[\![l]\!]_c = [\![r]\!]_c$, where we have defined these shorthands:

$$\begin{aligned} l &= \texttt{x := 0; } c \\ c &= \texttt{if iszero y then x := y else y := x} \\ r &= \texttt{x := 0; y := 0} \end{aligned}$$

To do this, we must show that $[\![l]\!]_c(\sigma_0) = [\![r]\!]_c(\sigma_0)$ for an arbitrary initial state $\sigma_0$.

In addition to properties (3), (4), and (5), we will need two further generic properties about state update:

$$([\texttt{x} \mapsto n]\sigma)(\texttt{y}) = \sigma(\texttt{y}) \text{ if } \texttt{x} \text{ and } \texttt{y} \text{ are different variables} \qquad (9)$$

$$([\texttt{y} \mapsto n_2]([\texttt{x} \mapsto n_1]\sigma) = [\texttt{x} \mapsto n_1]([\texttt{y} \mapsto n_2]\sigma) \text{ if } \texttt{x} \text{ and } \texttt{y} \text{ are different variables} \qquad (10)$$

First, note that for any $\sigma$ and any variable $v$,

$$[\![v \texttt{ := 0}]\!]_c(\sigma) = [v \mapsto [\![\texttt{0}]\!]_a(\sigma)]\sigma = [v \mapsto 0]\sigma \qquad (11)$$

Then on the right hand side we have

$$\begin{aligned}[\![r]\!]_c(\sigma_0) &= ([\![\texttt{y := 0}]\!]_c \circ_\perp [\![\texttt{x := 0}]\!]_c)(\sigma_0) \\ &= [\texttt{y} \mapsto 0]([\texttt{x} \mapsto 0]\sigma_0)\end{aligned} \tag{12}$$

On the left hand side we have

$$\begin{aligned}[\![l]\!]_c(\sigma_0) &= [\![\texttt{x := 0; } c]\!]_c(\sigma_0) \\ &= ([\![c]\!]_c \circ_\perp [\![\texttt{x := 0}]\!]_c)(\sigma_0) \\ &= [\![c]\!]_c([\texttt{x} \mapsto 0]\sigma_0) && \text{by (11)} \\ &= [\![c]\!]_c(\sigma_1) && \text{where } \sigma_1 = [\texttt{x} \mapsto 0]\sigma_0 \\ &= \mathit{if}\ [\![\texttt{iszero y}]\!]_b(\sigma_1)\ \mathit{then}\ [\![\texttt{x := y}]\!]_c(\sigma_1)\ \mathit{else}\ [\![\texttt{y := x}]\!]_c(\sigma_1) \\ &= \mathit{if}\ \sigma_1(\texttt{y}) = 0\ \mathit{then}\ [\texttt{x} \mapsto \sigma_1(\texttt{y})]\sigma_1\ \mathit{else}\ [\texttt{y} \mapsto \sigma_1(\texttt{x})]\sigma_1\end{aligned}$$

where the "*if-then-else*" notation is just shorthand for the two cases in the mathematical definition of $[\![]\!]_c$ on `if` commands.

Now, to continue the calculation, we separately consider two cases for the initial value of `y`:

- $\sigma_0(\texttt{y}) = 0$.

  Then, by (9), $\sigma_1(\texttt{y}) = 0$ as well. So

  $$\begin{aligned}[\![l]\!]_c(\sigma_0) &= \ldots \\ &= \mathit{if}\ \sigma_1(\texttt{y}) = 0\ \mathit{then}\ [\texttt{x} \mapsto \sigma_1(\texttt{y})]\sigma_1\ \mathit{else}\ [\texttt{y} \mapsto \sigma_1(\texttt{x})]\sigma_1 \\ &= [\texttt{x} \mapsto 0]\sigma_1 && \text{by (4)} \\ &= [\texttt{x} \mapsto 0]\sigma_0 \\ &= [\texttt{x} \mapsto 0]([\texttt{y} \mapsto 0]\sigma_0) && \text{by (5)} \\ &= [\texttt{y} \mapsto 0]([\texttt{x} \mapsto 0]\sigma_0) && \text{by (10)} \\ &= [\![r]\!]_c(\sigma_0) && \text{by (12)}\end{aligned}$$

- $\sigma_0(\texttt{y}) \neq 0$.

  Then, by (9), $\sigma_1(\texttt{y}) \neq 0$ as well. So

  $$\begin{aligned}[\![l]\!]_c(\sigma_0) &= \ldots \\ &= \mathit{if}\ \sigma_1(\texttt{y}) = 0\ \mathit{then}\ [\texttt{x} \mapsto \sigma_1(\texttt{y})]\sigma_1\ \mathit{else}\ [\texttt{y} \mapsto \sigma_1(\texttt{x})]\sigma_1 \\ &= [\texttt{y} \mapsto \sigma_1(\texttt{x})]\sigma_1 && \text{by (3)} \\ &= [\texttt{y} \mapsto 0]\sigma_1 \\ &= [\texttt{y} \mapsto 0]([\texttt{x} \mapsto 0]\sigma_0) \\ &= [\![l]\!]_c\sigma_0 && \text{by (12)}\end{aligned}$$

□

3. To handle `repeat` in the same style as `while`, we need to start with a recursive equality analogous to the one on page 6, namely:

$$[\![\texttt{repeat c}_1\ \texttt{until b}_2]\!]_c = [\![\texttt{c}_1\texttt{; if b}_2\ \texttt{then skip else repeat c}_1\ \texttt{until b}_2]\!]_c$$

Unfolding the denotation function on the right hand side, and writing $r$ for $[\![\texttt{repeat c}_1\ \texttt{until b}_2]\!]_c$, $b$ for $[\![\texttt{b}_2]\!]_c$ and $c$ for $[\![\texttt{c}_1]\!]_c$, we get

$$r(\sigma) = \mathit{if}\ b(c(\sigma))\ \mathit{then}\ c(\sigma)\ \mathit{else}\ r(c(\sigma))$$

The solution to this recursive equation is $\mathit{fix}(G)$, the least-fixed point of the functional $G$ given by

$$G = \lambda u \lambda \sigma.\mathit{if}\ b(c(\sigma))\ \mathit{then}\ c(\sigma)\ \mathit{else}\ u(c(\sigma))$$

The corresponding function on graphs, analogous to (8), is given by

$$\lceil G(u) \rceil = \{(\sigma, c(\sigma)) \mid b(c(\sigma))\} \cup \{(\sigma, \sigma') \mid \neg b(c(\sigma)) \wedge (c(\sigma), \sigma') \in \lceil u \rceil\} \qquad (8')$$

The approximations to $r$ are given by

$$\begin{array}{lcl} r_0 & = & \lambda\sigma.\bot \\ r_{i+1} & = & G(r_i) \\ & = & G^i(r_0) \end{array}$$

where $G^i = G \circ G \circ \cdots \circ G$, $i$ times.

Analogous to the `while` case, we claim that $\mathit{fix}(G)$ is given by

$$\lceil r \rceil = R = \bigcup_{i=0}^{\infty} \lceil G(r_i) \rceil = \bigcup_{i=0}^{\infty} \lceil G^i(r_0) \rceil$$

To show that this least fixed point exists, we must reprove the p. 11 Theorem for $r$ and $G$ in place of $w$ and $F$.

The only potential changes required to the proof are the places that mention (8); we must now make sure that they also work for (8′). First, we must prove

LEMMA. $G$ is monotonic.

PROOF Assume $\lceil u \rceil \subseteq \lceil u' \rceil$ and suppose $(\sigma, \sigma') \in \lceil G(u) \rceil$. Then by (8′), there are two cases. Case (i): $\sigma' = c(\sigma)$ and $b(c(\sigma))$ holds. Then, by (8′), $(\sigma, \sigma') \in \lceil G(u') \rceil$ too. Case (ii): $\neg b(c(\sigma))$ holds and $(c(\sigma), \sigma') \in \lceil u \rceil$. Then by the assumption, $(c(\sigma), \sigma') \in \lceil u' \rceil$, and by (8′), $(\sigma, \sigma') \in \lceil F(u') \rceil$. □

Second, we must prove that $\lceil G(r) \rceil \subseteq \lceil r \rceil$.

Consider any $(\sigma, \sigma') \in \lceil G(r) \rceil$. By (8′), there are two cases. Case (i): $\sigma' = c(\sigma)$ and $b(c(\sigma))$ holds. Then $(\sigma, \sigma') \in \lceil r_1 \rceil = \lceil G(r_0) \rceil \subseteq \bigcup_{i=0}^{\infty} \lceil G(r_i) \rceil = \lceil r \rceil$. Case (ii): $\neg b(c(\sigma))$ holds and $(c(\sigma), \sigma') \in \lceil r \rceil = \bigcup_{i=0}^{\infty} \lceil r_i \rceil$. So there must be some (finite) $n$ such that $(c(\sigma), \sigma') \in \lceil r_n \rceil$. So by (8′), $(\sigma, \sigma') \in \lceil G(r_n) \rceil \subseteq \bigcup_{i=0}^{\infty} \lceil G(r_i) \rceil = \lceil r \rceil$. □