Connecting semantics for While to that for PCF

Andrea Schalk
A.Schalk@manchester.ac.uk

6th May 2023

The aim of these notes is to create a connection between the standard model of the While programming language as, for example, described in [NN07], and the standard semantics for PCF, as for example given in [Str06]. For the latter we make copious reference to [RS22], adopting the notation given there and also using specific results. We assume the reader is familiar with the syntax and operational semantics of While.

In both cases fixed point operators are at work, and we want to see here what the connection between the two may be.

I would like to thank Vlad Sirbu and Joe Razavi for their helpful comments and ideas.

1 Interpreting While programs

We give a brief introduction into the first language at issue and describe a denotational semantics which is a very mild variation on that given in [NN07].

1.1 The While language

The While language is a theoretical imperative language that allows us to study key aspects of that paradigm. At the University of Manchester it is the vehicle for introducing students to ideas such as computability, decidability, and a 'natural semantics' in the form of a transition system that tracks how the values of variables change as the program is executed. Students are also given a first account of reasoning about programs via Hoare triples in a way that matches the transition system.

The language is based on the idea of a program where steps (known as *statements*) are carried out one at a time making it natural to ask for a *compositional* interpretation. The constructs available are assigning values in the form of natural numbers to variables, a conditional statement, and loops in the form of 'while' loops. For theoretical reasons there is also a statement `skip` which does not have any effect. In order to support values which may be assigned to variables (and to carry out computations on these) and the conditionals the language has a notion of *arithmetic* and *boolean expressions*, which have obvious interpretations in the sets $\mathbb{N}$ and $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$.

1.2 Interpreting While programs: states

As one might expect for an imperative language, the effect of a program is tracked by how the values assigned to the available variables change as the program is executed.

We assume that we have a supply of variables Vars. It is standard to demand that the supply of variables is inexhaustible, but, of course, any one program can only mention finitely many variables, and we see below that neither the transitional nor the denotational semantics rely on variables that are not mentioned in the given program. Nielson and Nielson in [NN07] assume

that we might as well work under the principle that we have a value for every variable in Vars available to us; they then adopt the convention that if we give a finite description, say

$$[x \mapsto 3,\ y \mapsto 0]$$

we assume that the value for each of the remaining variables in Vars is 0. We follow their lead.

A **state** is a function from Vars to $\mathbb{N}$. We may **update a state** by changing the value of one of the variables, and we use the notation

$$\sigma[x \mapsto n]$$

to describe the function that is given by the assignment

$$y \longmapsto \begin{cases} n & y = x \\ \sigma y & \text{else.} \end{cases}$$

Where we carry out several updates in succession we assume that we do so from left to right.

We may think of the set of all states as $\mathsf{St} = \mathsf{Fun}(\mathsf{Vars}, \mathbb{N})$, but this is not always the most suitable manifestation of the idea. We may alternatively think of it as an infinite produce of the form

$$\prod_{x \in \mathsf{Vars}} \mathbb{N} = \{(n_x)_{x \in \mathsf{Vars}} \mid n_x \in \mathbb{N}\},$$

that is an (infinite) tupel which has one coordinate for each variable. We say that the product (or the tuples) is *indexed by the elements of the set* Vars.

There are bijections between the two sets, namely we map a function $\sigma : \mathsf{Vars} \longrightarrow \mathbb{N}$ to the tupel

$$(\sigma x)_{x \in \mathsf{Vars}},$$

that is, the tupel which, at the component for the variable x, has the value σx assigned to x in the state σ.

In the opposite direction, given a tupel $(n_x)_{x \in \mathsf{Vars}}$, we may define a function from Vars to $\mathbb{N}$ by the assignment

$$x \longmapsto n_x.$$

We allow ourselves to switch our point of view of states between these two as we find convenient.

We sometimes want to restrict a given state to only assign values to some of the variables, and we employ the machinery described below to do so.

For every variable x we get a function from St to $\mathbb{N}$, which we may think of in two ways depending on which point of view we have in mind. We set

$$\begin{aligned} \mathsf{ev}^x : \ \mathsf{Fun}(\mathsf{Vars}, \mathbb{N}) &\longrightarrow \mathbb{N} \\ \sigma &\longmapsto \sigma x, \end{aligned}$$

and

$$\begin{aligned} \pi_x : \ \prod_{y \in \mathsf{Vars}} \mathbb{N} &\longrightarrow \mathbb{N} \\ (n_y)_{y \in \mathsf{Vars}} &\longmapsto n_x. \end{aligned}$$

If we start with an element σ of $\mathsf{Fun}(\mathsf{Vars}, \mathbb{N})$, apply our isomorphism to obtain an element of $\prod_{x \in \mathsf{Vars}} \mathbb{N}$ and then apply the function π_x we get the same result as if we apply ev^x to σ.

If we are interested in the values of several variables, say $x_1, x_2, \ldots, x_n$ we may define a function that is more easily viewed as

$$\pi_{x_1,\ldots,x_n} : \ \prod_{x \in \mathsf{Vars}} \mathbb{N} \longrightarrow \prod_{1 \leq i \leq n} \mathbb{N}$$

which maps an infinite tupel to the n-tupel that only retains the values of the chosen variables x_1, x_2 to x_n.

On the other hand, if we have values for some variables, we may define a state that assigns those values to the given variables, and 0 to all the variables not mentioned. This provides us with a function

$$\mathrm{zr}_{x_1,\dots,x_n} : \prod_{1\le i\le n} \mathbb{N} \longrightarrow \prod_{x\in \mathsf{Vars}} \mathbb{N}$$

given by the fact that the component for x of $\mathrm{zr}_{x_1,\dots,x_n}(m_1,\dots,m_n)$ is given by

$$\Big(\mathrm{zr}_{x_1,\dots,x_n}(m_1,\dots,m_n)\Big)_x = \begin{cases} m_i & x = x_i \\ 0 & \text{else.} \end{cases}$$

We have the following equalities connecting these functions:

$$\pi_{x_1,\dots,x_n} \circ \mathrm{zr}_{x_1,\dots,x_n} = \mathrm{id}_{\prod_{1\le i\le n}\mathbb{N}},$$

while the yth component of $\mathrm{zr}_{x_1,\dots,x_n}\, \pi_{x_1,\dots,x_n}(n_x)_{x\in\mathsf{Vars}}$ agrees with that of the input $(n_x)_{x\in x_1,\dots,x_n}$ provided that y is one of the variables x_1, x_2,…, x_n. In general we can't say anything about whether their respective values for any of the other variables are equal.

1.3 Interpreting While programs: Typing

Our intention is to interpret a While program as a function that manipulates states. However, the program might not terminate, in which case there is nothing safe we can say about the resulting change to the start state.

There are two ways of modelling this situation:

- We may describe the interpretation of a program as a partial function on the set of states St or
- we may describe the interpretation of a program as a function that outputs an element of the set $\mathsf{St}_\perp = \mathsf{St} \cup \{\perp\}$, where we add an additional element $\perp$ to the set of states to signify that there is no 'output state' in some situations.

The former solution is chosen by Nielson and Nielson while we prefer the latter due to the fact that it connects more easily with our semantics for PCF. However, this choice is in some sense immaterial since the two settings for the denotational semantics are isomorphic to each other, see Proposition 1.1.

When we consider the elements of $\mathsf{St}_\perp$ we use a variable τ, but we frequently want to emphasize that we know our element to be a state, rather than $\perp$, and in that situation we use the variable σ. Further when we talk about elements of this set we typically make case distinctions based on whether the element in question is $\perp$ or not, and we typically treat that case first, and subsequently assume that we have a state, which we may query for the value it assigns to some variable.

We want the semantics to be *compositional* in the sense that given a program P and a statement S we expect the interpretation of the program P ; S to be the composite of the interpretation of S with that of P, that is

$$[\![P\,;S]\!] = [\![S]\!] \circ [\![P]\!].$$

Consequently we have to ensure that the interpretation of a program has the same source and target set, and so for us the interpretation of a program is a function from $\mathsf{St}_\perp$ to itself. However, we do not want to view $\mathsf{St}_\perp$ merely as a set, but as a *partially ordered set.*

For a set S, when we form $S_\perp$ we equip it with the partial order for which $\perp \leq s$ for all $s \in S$ are the only non-reflexive instances of the order relation. This is a *flat partial order* (see Section 4.3 of [RS22]).

When we consider $\mathsf{St}_\perp$ in this way we have an element $\perp$ which we may consider as 'less informative' than all the states. The element $\perp$ is the least element of $\mathsf{St}_\perp$. We say that an order-preserving function between two posets with least elements is **strict** if it maps the least element of the source set to the least element of the target set.

In Section 1.5 we give a formal definition of the interpretation of a While program P as a strict (and so order-preserving) function

$$[\![P]\!] : \mathsf{St}_\perp \longrightarrow \mathsf{St}_\perp.$$

We further note that the effect of running the program P on the state σ has the consequence of updating the state σ accordingly, *provided that P terminates*, and if the variables that occur in P are $x_1, \ldots, x_n$ we would expect our semantics to be such that the effect of that is

$$\sigma[x_1 \mapsto ([\![P]\!]\sigma)x_1] \cdots [x_n \mapsto ([\![P]\!]\sigma)x_n].$$

This is established in Proposition 1.4.

We provide a result that tells us what we need to check to ensure that a function is suitable to interpret a While program.

Proposition 1.1

The following statements hold.

(i) A strict function from $S_\perp$ to $T_\perp$ is order-preserving.

(ii) A function from $S_\perp$ to $T_\perp$ is order-preserving if and only if

- the function is strict or
- the function is constant.

Proof. Routine.

For posets P and Q with least elements we write

$$P \Rightarrow_{\mathsf{s}} Q$$

for the partially ordered set of strict order-preserving functions from P to Q with the element-wise partial order.

We interpret a While program P as an element of $\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$.

When interpreting While programs we only rely on fixed points to exist in the partially ordered set

$$(\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp) \Rightarrow (\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp)$$

of Scott-continuous functions from $\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$ to itself in order to give a meaning to loops.

It is important to realize that $\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$ is *not a poset with finite height*, even if we only have one variable. For example, we may find an infinite chain of functions from the constant function $k_\perp$, which produces the output $\perp$ for every input, to the identity function on $\mathsf{St}_\perp$: Let the only variable be x, and $f_n : \mathsf{St}_\perp \longrightarrow \mathsf{St}_\perp$ be given by the assignment which

- maps $\perp$ to $\perp$,

- for $1 \leq i \leq n$ it maps each state $[x \mapsto i]$ to itself and
- for $i > n$ it maps the state $[x \mapsto i]$ to $\bot$.

Then $k_\bot < f_1 < f_2 \cdots < \mathrm{id}_{\mathsf{St}_\bot}$.

Hence the fixed points employed to interpret while loops (see Section 1.5) are non-trivial.

As is the case for the semantics for PCF described in [RS22], the least fixed point of a Scott-continuous function

$$F : \mathsf{St}_\bot \Rightarrow_s \mathsf{St}_\bot \longrightarrow \mathsf{St}_\bot \Rightarrow_s \mathsf{St}_\bot$$

may be calculated as

$$\bigvee_{j \in \mathbb{N}} F^j k_\bot,$$

where again $k_\bot$ is the function that assigns $\bot$ to each input from $\mathsf{St}_\bot$.

In Section 1.6 we see this construction in action, and in particular we see that for simple examples, we only have to compute repeated applications of F to $k_\bot$ a small number of times. The following result helps us understand how we may calculate directed suprema. in $\mathsf{St}_\bot \Rightarrow_s \mathsf{St}_\bot$.

Proposition 1.2

Let $\mathcal{F}$ be a directed set of strict functions from $\mathsf{St}_\bot$ to itself. Given a state σ it is the case that

(i) if there exists $f \in \mathcal{F}$ with $(f\sigma) \neq \bot$ then for all $f' \in \mathcal{F}$ with $f \leq f'$ it is the case that $f\sigma = f'\sigma$ and

(ii)
$$\left(\bigvee \mathcal{F}\right)\sigma = \bigvee_{f \in \mathcal{F}} f\sigma = \begin{cases} f\sigma & \exists f \in \mathcal{F} \,.\, f\sigma \neq \bot \\ \bot & \text{else.} \end{cases}$$

Proof. The first statement is an immediate consequence of the partial order on $\mathsf{St}_\bot$: We know that strict functions preserve the order, and that for all f, f' and σ as given it must be the case that $f\sigma \leq f'\sigma$, but the only non-trivial instances of the partial order in $\mathsf{St}_\bot$ are $\bot \leq \sigma'$, and so if $f\sigma \neq \bot$ then $f\sigma = f'\sigma$ must hold.

The second statement follows from the first, including the fact that the expression on the right does not depend on the choice of f.

We conclude this section by making the connection with the semantics described in [NN07]. The authors choose the set of partial functions from St to itself to interpret While programs, and they equip this set with the partial order given by

$$f \preceq g \qquad \text{if and only if} \qquad \text{for all } \sigma \in \mathsf{St} \quad f\sigma \text{ defined} \implies g\sigma \text{ defined and } f\sigma = g\sigma.$$

In other words, g is 'more defined' than f, but takes on the same values as f for elements of the domain of definition of f. This partial order is required to ensure the existence of the fixed points required to interpret loops.

It turns out that their setting is not substantially different from ours:

Proposition 1.3

The set of partial functions from S to T with the partial order given by $\preceq$ is isomorphic to the set of strict functions from $S_\perp$ to $T_\perp$ with the pointwise order.

Proof. Routine.

Given this we do occasionally refer to results in [NN07] translated into our description of the semantic setting.

1.4 Interpreting expressions

Our language comes with two kinds of expressions, arithmetic and boolean ones. Given a state we may interpret the former by natural numbers and the latter by an element of $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ via a simple recursive definition using the obvious operations on these two sets to interpret the various formation rules.

For an arithmetic expression a we may think of $[\![a]\!]$ as a function from St to $\mathbb{N}$. We do not need a bottom state in this situation since the meaning of an arithmetic expression is always well-defined. So we may think of $[\![a]\!]\sigma$ as the natural number given by the arithmetic expression a for the state σ.

Similarly, for a boolean expression we have a function $[\![b]\!]$ from St to $\mathbb{B}$.

1.5 Interpreting programs

For the simple statements we may use the following interpretations:

- The **`skip`** command takes no action, and so it should not be surprising that we set $[\![\mathtt{skip}]\!] = \mathrm{id}_{\mathsf{St}_\perp}$. We note that the identity function is strict.
- Updating the value of a particular variable also has a fairly obvious interpretation.

$$[\![x := a]\!]\tau = \begin{cases} \perp & \tau = \perp \\ \tau[x \mapsto [\![a]\!]\tau] & \text{else.} \end{cases}$$

  We abuse notation slightly here in that $[\![a]\!]\tau$ is only defined if τ is a state (rather than $\perp$), and we have left that implicit in our case distinction. We write case distinctions to first cover the case where the input is $\perp$, and we then assume that the input is a state for the remaining cases. Similar considerations apply to boolean expressions, see below. For the given definition it is obviously the case that the resulting function on states is strict.

As indicated in Section 1.3 we expect our interpretation to be compositional in the sense that if P is a program and S a statement we set

$$[\![P\,;S]\!] = [\![S]\!] \circ [\![P]\!].$$

The composition of two strict functions is strict.

For the remaining two formation rules we need to worry about what to do about the condition that appears.

We note that we have the following function

$$\mathrm{cond} : (\mathsf{St} \Rightarrow \mathbb{B}) \times (\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp) \times (\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp) \longrightarrow \mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$$

given by

$$\mathrm{cond}(b, f, f')\tau = \begin{cases} \bot & \tau = \bot \\ f\tau & b\tau = \mathtt{tt} \\ f'\tau & b\tau = \mathtt{ff}, \end{cases}$$

where

$$f, f' : \mathsf{St}_\bot \longrightarrow \mathsf{St}_\bot.$$

We again abuse notation slightly here: for the boolean b, $b\tau$ is defined only if τ is a state rather than $\bot$, and this is implicitly ruled out by the way we formulate the case distinction. We do this routinely whenever we consider this conditional function below.

We note that the function $\mathrm{cond}(b, g, g')$ is indeed strict, and so is suitable as an interpretation of a While program, as well as being Scott-continuous.

We may now define the interpretation of a conditional as

$$[\![\texttt{if } b \texttt{ then } P \texttt{ else } P']\!] = \mathrm{cond}([\![b]\!], [\![P]\!], [\![P']\!]).$$

This takes us to the interpretation of the looping construction.

We set

$$[\![\texttt{while } b \texttt{ do } P]\!] = \mathsf{fix}\, F,$$

where

$$\begin{array}{rcl} F : \mathsf{St}_\bot \Rightarrow_{\mathsf{s}} \mathsf{St}_\bot & \longrightarrow & \mathsf{St}_\bot \Rightarrow_{\mathsf{s}} \mathsf{St}_\bot \\ f & \longmapsto & \mathrm{cond}([\![b]\!], f \circ [\![P]\!], \mathrm{id}_{\mathsf{St}_\bot}) \end{array}.$$

We want to invoke Proposition 4.13 from [RS22] to ensure that this fixed point exists, for which we need to establish that this assignment is Scott-continuous in the argument g. If $\mathcal{F}$ is a directed subset of $\mathsf{St}_\bot \Rightarrow_{\mathsf{s}} \mathsf{St}_\bot$ then

$$\begin{array}{rll} F(\bigvee^{\uparrow} \mathcal{F}) = \mathrm{cond}([\![b]\!], (\bigvee^{\uparrow} \mathcal{F}) \circ [\![P]\!], \mathrm{id}_{\mathsf{St}_\bot}) & & \text{def } F \\ \quad = \mathrm{cond}([\![b]\!], \bigvee\limits_{f \in \mathcal{F}}^{\uparrow} (f \circ [\![P]\!]), \mathrm{id}_{\mathsf{St}_\bot}) & & \text{composition is Scott-continuous,} \end{array}$$

where we have to use a well-known result for dcpos in the final step. If we can show that

$$\bigvee_{f \in \mathcal{F}}^{\uparrow} Ff = \bigvee_{f \in \mathcal{F}}^{\uparrow} \mathrm{cond}([\![b]\!], f \circ [\![P]\!], \mathrm{id}_{\mathsf{St}_\bot}) = \mathrm{cond}([\![b]\!], \bigvee_{f \in \mathcal{F}}^{\uparrow} (f \circ [\![P]\!]), \mathrm{id}_{\mathsf{St}_\bot})$$

we are done. Assume that $\tau \in \mathsf{St}_\bot$ is a suitable input to this function. We have three cases:

- If $\tau = \bot$ then both expressions evaluate to $\bot$.
- If $[\![b]\!]\tau = \mathtt{tt}$ then

$$\begin{array}{rll} \mathrm{cond}([\![b]\!], \bigvee\limits_{f \in \mathcal{F}}^{\uparrow} (f \circ [\![P]\!]), \mathrm{id}_{\mathsf{St}_\bot})\tau = \bigvee\limits_{f \in \mathcal{F}}^{\uparrow} (f \circ [\![P]\!])\tau & & \text{def cond} \\ \quad = \left(\bigvee\limits_{f \in \mathcal{F}}^{\uparrow} \mathrm{cond}(b, f \circ [\![P]\!], \mathrm{id}_{\mathsf{St}_\bot}) \right) \tau & & \text{def cond} \end{array}$$

  as required.
- If $[\![b]\!]\tau = \mathtt{ff}$ then both expressions evaluate to τ.

Hence in all three cases the two functions agree, so we have

$$F\left(\bigvee^{\uparrow}\mathcal{F}\right) = \bigvee_{f\in\mathcal{F}}^{\uparrow} Ff,$$

which means that F is Scott-continuous as required.

It is now possible to show that our semantics has the effect foreshadowed at the end of Section 1.3. We further note that a program cannot affect the values of variables that it does not mention, and that its effect only depends on the values of variables that it does.

Proposition 1.4

If the variables that occur in the program P are among $x_1, x_2, \ldots x_n$ and σ is a state then

$$[\![P]\!]\sigma = \begin{cases} \sigma[x_1 \mapsto ([\![P]\!]\sigma)x_1]\cdots[x_n \mapsto ([\![P]\!]\sigma)x_n] & P \text{ terminates} \\ \bot & \text{else.} \end{cases}$$

If y is a variable that is not mentioned in P and P terminates when run on the state σ we have

$$([\![P]\!]\sigma)y = \sigma y.$$

Further if σ' is a state with $\sigma x_i = \sigma' x_i$ for $1 \leq i \leq n$ then either $[\![P]\!]\sigma = \bot = [\![P]\!]\sigma'$ or for all $1 \leq i \leq n$ we have $([\![P]\!]\sigma)x_i = ([\![P]\!]\sigma')x_i$.

Proof. For the first claim there are two points at issue: One may show by induction that the denotation of P has the following property: If P terminates for the state σ and $[\![P]\!]\sigma x \neq \sigma x$ then x must be mentioned in P—in other words, executing P can only change the value of variables that are mentioned in P. The base cases are given by the interpretation of `skip` and the assignment, for which this is straightforward to verify. For the three step cases it is a straightforward application of the induction hypothesis for composition and for the conditional. For loops we may show this by first proving that it is true for the function F whose fixed point is taken, provided it is true for the function f that appears as its argument, and then observing that

- it is true for the function $k_\bot$ which is the 'seed ' for calculating the fixed point and
- if $\mathcal{F}$ is a directed set of strict endofunctions on $\mathsf{St}_\bot$ which do not affect the values of variables not mentioned in P then this is also true for their directed supremum since

  $$\left(\bigvee^{\uparrow}\mathcal{F}\right)\sigma = \bigvee_{f\in\mathcal{F}}^{\uparrow} f\sigma = \begin{cases} f\sigma & \exists f \in \mathcal{F}\,.\, f\sigma \neq \bot \\ \bot & \text{else} \end{cases}$$

  by Proposition 1.2, and this shows that the directed supremum $\bigvee^{\uparrow}\mathcal{F}$ may only affect variables that are affected by one of the $f \in \mathcal{F}$.

The second point at issue is that of non-termination versus termination. In order to define the notion of non-termination for a program we must refer to the structural operational semantics as given in [NN07], which is there shown to give the same result as our denotational semantics (and which my be transferred to our order-isomorphic setting). From this result we know that for a state σ it is the case that

$$[\![P]\!]\sigma = \bot$$

if and only if

P does not terminate when started in state σ.

The second statement is an immediate consequence of the first by ensuring that y is not among the x_i.

The final claim may also be shown by induction over the definition of the denotation. One first has to show that the corresponding statement holds for the interpretation of expressions, where it is a simple observation. For programs, in the base cases, this is trivial for the **skip** statement, and for the update statement it follows from the fact that the statement holds for expressions. Again, the step cases of composition and conditional are simple applications of the induction hypothesis, and it is only the case of loops where one has to look a bit deeper. The proof is then much the same as that for the first statement: It's true for the constant function $k_\perp$, and if we have a directed set of function $\mathcal{F}$ of strict functions from $\mathsf{St}_\perp$ to itself such that for each $f \in \mathcal{F}$ we have $f\sigma = f\sigma'$ then this is also true for $\bigvee \mathcal{F}$.

We may use our functions π and zr from Section 1.2 to reformulate some of the statements from the previous result:

If y is not mentioned in P then

$$\pi_y \circ [\![P]\!] = \pi_y.$$

If $x_1, x_2, \ldots, x_n$ contain all the variables mentioned in P then for states σ and σ' we have

$$\pi_{x_1,\ldots,x_n}\sigma = \pi_{x_1,\ldots,x_n}\sigma' \qquad \text{implies} \qquad \pi_{x_1,\ldots,x_n}([\![P]\!]\sigma) = \pi_{x_1,\ldots,x_n}([\![P]\!]\sigma').$$

In particular

$$\pi_{x_1,\ldots,x_n} \circ [\![P]\!] = \pi_{x_1,\ldots,x_n} \circ [\![P]\!] \circ \mathrm{zr}_{x_1,\ldots,x_n} \circ \pi_{x_1,\ldots,x_n}$$

and

$$\mathrm{zr}_{x_1,\ldots,x_n} \circ \pi_{x_1,\ldots,x_n} \circ [\![P]\!] \circ \mathrm{zr}_{x_1,\ldots,x_n} = [\![P]\!] \circ \mathrm{zr}_{x_1,\ldots,x_n}. \qquad (\dagger)$$

This result indicates that one might define a variation of the given semantics, where instead of demanding that a state provides values for all variables, we only need to ensure that it gives one for all variables mentioned in P. We make use of related ideas in Section 3.

We may now examine the definition of the denotational semantics to determine what might cause non-termination: It is clear that the two statements terminate for all states, and that if P and P' are programs that terminate for the state σ then so are $P\,;P'$ and **if** b **then** P **else** P'. Hence as expected the only possible cause of non-termination is a **while** loop.

1.6 Examples

It may be helpful to show some very simple examples of the fixed point operator in action.

Non-terminating loop

The interpretation of

```
while true do skip
```

is the fixed point of the function $F : \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp \longrightarrow \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp$ given by

$$Fg\tau = \begin{cases} \perp & \tau = \perp \\ f\tau & \text{else.} \end{cases}$$

We further know that we may compute the fixed point by repeatedly applying F to the constant function $k_\perp$. We may see that if we apply F to $k_\perp$ we get $k_\perp$. Indeed, the given program does not terminate, so no matter in what state we start running it, the output should be the bottom element of $\mathsf{St}_\perp$.

Terminating loop without effect

We next look at

```
while false do skip.
```

Its interpretation is the fixed point of the function $F : \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp \longrightarrow \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp$ given by

$$Ff\tau = \begin{cases} \perp & \tau = \perp \\ \tau & \text{else.} \end{cases}$$

We may calculate that

$$Fk_\perp\tau = \begin{cases} \perp & \tau = \perp \\ \tau & \text{else,} \end{cases}$$

and it is easy to check that if we apply F again we obtain the same result, so this is a fixed point. This is, in fact, the identity function on $\mathsf{St}_\perp$ and we would indeed expect that running the above program has no effect on the state in which we start.

Simple terminating loop with limited effect

For something mildly more interesting, consider

```
while ~(x=1) do x:=1.
```

Its interpretation is $\mathsf{fix}\, F$ where $F : \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp \longrightarrow \mathsf{St}_\perp \Rightarrow \mathsf{St}_\perp$ given by

$$Ff\tau = \begin{cases} \perp & \tau = \perp \\ f(\tau[x \mapsto 1]) & \tau x \neq 1 \\ \tau & \text{else.} \end{cases}$$

We may calculate that

$$Fk_\perp\tau = \begin{cases} \perp & \tau = \perp \text{ or } \tau x \neq 1 \\ \tau & \text{else,} \end{cases}$$

and if we apply F again we get

$$FFk_\perp\tau = \begin{cases} \perp & \tau = \perp \\ Fk_\perp(\tau[x \mapsto 1]) & \tau x \neq 1 \\ \tau & \text{else,} \end{cases} = \begin{cases} \perp & \tau = \perp \\ \tau[x \mapsto 1] & \tau x \neq 1 \\ \tau & \text{else} \end{cases} = \begin{cases} \perp & \tau = \perp \\ \tau[x \mapsto 1] & \text{else.} \end{cases}$$

which is a fixed point. Once again the fixed point models the expected effect of our program on the start state.

Loop with varying termination behaviour

We finally look at a situation where the loop sometimes terminates, and sometimes doesn't, depending on the state we start in.

```
while ~(x=1) do x:=y
```

The function F whose fixed point gives the denotation of this program is given by

$$Ff\tau = \begin{cases} \bot & \tau = \bot \\ f(\tau[x \mapsto \tau y]) & \tau x \neq 1 \\ \tau & \tau x = 1. \end{cases}$$

We calculate that

$$Fk_\bot\tau = \begin{cases} \bot & \tau = \bot \text{ or } \tau x \neq 1 \\ \tau & \tau x = 1, \end{cases}$$

and if we apply F again we have

$$\begin{aligned} FFk_\bot\tau &= \begin{cases} \bot & \tau = \bot \\ Fk_\bot(\tau[x \mapsto \tau y]) & \tau x \neq 1 \\ \tau & \tau x = 1 \end{cases} \\ &= \begin{cases} \bot & \tau = \bot \text{ or } (\tau x \neq 1 \text{ and } \tau y \neq 1) \\ \tau[x \mapsto \tau y] & \tau x \neq 1 \text{ and } \tau y = 1 \\ \tau & \tau x = 1 \end{cases} \\ &= \begin{cases} \bot & \tau = \bot \text{ or } (\tau x \neq 1 \text{ and } \tau y \neq 1) \\ \tau[x \mapsto 1] & \tau x \neq 1 \text{ and } \tau y = 1 \\ \tau & \tau x = 1. \end{cases} \end{aligned}$$

This is a fixed point for F, and so the interpretation of the given program. We make the following observations:

- While we are able to describe the various case distinctions based on the given input, the case distinctions may become complicated for an arbitrary while loop.
- In the general case the number of case distinctions may increase in a situation where the number of times the body of the loop is executed varies depending on the state where the program starts, or where there are further conditionals involved.
- Consequently there is no simple way of calculating output of the fixed point for a given input (other than $\bot$).

2 Translating While programs to PCF

My student Vlad Sirbu [Sir22] has defined with a translation from While programs to PCF.

2.1 Idea

The underlying idea is that variables that appear in the While program P will appear as *free variables* in the PCF translation TP, and we may then interpret a state σ as a valuation that provides suitable values for those variables.

Since a state assigns a value to *all* available variables, it will typically provide values for variables that do not appear in TP, but that is fine since Proposition 2.36 of [RS22] ensures that this does not affect the denotation of TP. Further, Proposition 1.4 tells us that the interpretation of P does not change if we manipulate a given state by changing the values of variables that don't occur in P, so our two notions of interpretation, for TP and P, match there.

Since the only free variables that appear in terms of the form TP are of type nat we can do without a type environment here, and just assume that free variables in our terms are all of that type. This makes the notation a little less complicated.

A typical While program may potentially change any one variable in Vars, and there is no designated output variable. On the other hand, since we do not have product types in PCF, we can't easily mimic the effect of the given program on all its variables, but instead we have to pick one of the available variables as our 'output'.

Hence the translation has some parameters, namely

- a (finite) set of variables $D = \{x_1, x_2, \dots, x_n\}$ which contains all the variables that occur in P and
- the chosen output variable x_i.

What should we expect from a valid translation? We aim to compare the original program P with its translation $T^D_{x_i}\, P$ via the denotations for the two language, but we need to take a little extra care here.

$[\![P]\!]$ is a strict function from $\mathsf{St}_\perp$ to itself, and we would like to know the value of a $[\![P]\!]\sigma$ at the declared output variable x_i. If $[\![P]\!]\sigma$ is a state, then we may apply it to x_i to obtain an element of $\mathbb{N}$, but if $[\![P]\!]\sigma = \perp$, then this is not a priori an expression we may apply to x_i.

We therefore now think of each element of $\mathsf{St}_\perp$ as providing us with a function from Vars to $\mathbb{N}_\perp$ where we set

$$\tau x = \begin{cases} \perp & \tau = \perp \\ \tau x & \text{else.} \end{cases}$$

In Section 1.2, given a variable X we define a function $\mathrm{ev}^x : \mathsf{St} \longrightarrow \mathbb{N}$ which returns the value of the variable x in the given input state. We may extend this function to a function

$$\mathrm{ev}^x_\perp : \mathsf{St}_\perp \longrightarrow \mathbb{N}_\perp$$

by mapping an element $\tau \in \mathsf{St}_\perp$ to τx using this idea.

In other words we view $\mathsf{St}_\perp$ as a sub-poset of $\mathsf{Vars} \Rightarrow \mathbb{N}_\perp$, where we equip Vars with the discrete partial order. Note that when we apply a state σ to a variable we are always guaranteed an element of $\mathbb{N}$, it is only when we apply $\perp \in \mathsf{St}_\perp$ to a variable that we obtain the value $\perp \in \mathbb{N}_\perp$. Hence $\mathsf{Vars} \Rightarrow \mathbb{N}_\perp$ has many elements that do not correspond to elements of $\mathsf{St}_\perp$: Any function that assigns $\perp$ to a natural number, but doesn't do so for all of them, is such an element.

Having explained this usage we may now formulate a property that indicates that our translation works properly, namely we want for every state σ that

$$([\![P]\!]\sigma)x_i = [\![T^D_{x_i}\, P]\!]_\sigma. \qquad (*)$$

In other words when P has run for start state σ, the value for the variable x_i is equal to the value of the translation of P for output variable x_i, provided that the original values of the variables in D are provided by σ.

Here is a diagram to explain the idea more abstractly:

$$\begin{array}{ccc} \mathsf{St} & \xrightarrow{[\![T^D_{x_i}\, P]\!]_\sigma} & \mathbb{N}_\perp \\ {\scriptstyle [\![P]\!]|_{\mathsf{St}}}\downarrow & & \| \\ \mathsf{St}_\perp & \xrightarrow[\mathrm{ev}^{x_i}]{} & \mathbb{N}_\perp \end{array}$$

where we restrict the denotation of P to states and evaluate the given state at the variable x_i along the left and bottom, while along the top we have the denotation of the translated term which provides an element of $\mathbb{N}_\perp$.

2.2 The translation

For expressions we do not have to worry about output variables, and we abuse notation by reusing T for our translation of these:

For an arithmetic expressions we get a PCF term Ta by demonstrating that we may define PCF terms for each of the arithmetic operations that are permitted in While so that for every state σ we have

$$[\![a]\!]\,\sigma = [\![Ta]\!]_\sigma.$$

For boolean expressions, we need to match the conditional available in PCF, which is for natural numbers rather then for booleans. We use the function

$$\begin{aligned} I : \mathbb{B} &\longrightarrow \mathbb{N} \\ x &\longmapsto \begin{cases} 0 & x = \mathsf{tt} \\ 1 & x = \mathsf{ff} \end{cases} \end{aligned}$$

for this purpose. We may then provide appropriate PCF terms for each of the boolean operations available in While to give our translation the property that

$$I\,[\![b]\!]\,\sigma = [\![Tb]\!]_\sigma.$$

In [Sir22] we find most of a proof by induction that the property $(*)$ holds, requiring only the properties of PCF as given in [RS22], and the denotation for While programs as summarized above. There is one gap when it comes to the looping construction, and one aim of these notes is to provide the result on denotations that allows this gap to be filled.

To convey the flavour of the translation and the process of establishing property $(*)$, we provide a sketch of the proof. We assume that P is a While program and that $D = \{x_1, x_2, \ldots, x_n\}$ contains the variables that occur in P.

Note that in order to treat non-termination of P appropriately we have to insert additional conditionals into the PCF term, where both branches are the same, and we use the following notational shortcut in that situation: Let t and t' be PCF terms of type nat, then

$$\mathsf{ifz}\ t\,[t']^2 = \mathsf{ifz}\,t[t'][t'].$$

If we abstract over a number of variables of the same type, we abbreviate as follows:

$$\lambda x_1 \cdots x_n : \mathsf{nat}.\, t = \lambda x_1 : \mathsf{nat}.\, \lambda x_2 : \mathsf{nat}.\, \cdots \lambda x_n : \mathsf{nat}.\, t.$$

- For the **`skip`** statement we have

$$[\![T^D_{x_i}\ \mathtt{skip}]\!]_\sigma = [\![x_i]\!]_\sigma = \sigma x_i = ([\![\mathtt{skip}]\!]\,\sigma)x_i.$$

- For an assignment we have

$$[\![T^D_{x_i}(x := a)]\!]_\sigma = [\![(\lambda x : \mathsf{nat}.\, x_i) T a]\!]_\sigma = \begin{cases} [\![T a]\!]_\sigma & x = x_i \\ [\![x_i]\!]_\sigma = \sigma x_i & \text{else,} \end{cases}$$

  while

$$([\![x := a]\!]\sigma) x_i = (\sigma[x \mapsto [\![a]\!]\sigma])\, x_i = \begin{cases} [\![a]\!]\sigma & x = x_i \\ \sigma x_i & \text{else} \end{cases}$$

  as required if we invoke the property of the translation of arithmetic expressions from above.

- For the conditional we may reason as follows: We have

$$\begin{aligned} [\![T^D_{x_i}(\texttt{if}\, b\ \texttt{then}\, P\ \texttt{else}\, P')]\!]_\sigma &= [\![\mathsf{ifz}\,(T b)[T^D_{x_i} P][T^D_{x_i} P']]\!]_\sigma \\ &= \mathrm{ifz}\, [\![T b]\!]_\sigma [\![T^D_{x_i} P]\!]_\sigma [\![T^D_{x_i} P']\!]_\sigma \\ &= \begin{cases} [\![T^D_{x_i} P]\!]_\sigma & [\![T b]\!]_\sigma = 0 \\ [\![T^D_{x_i} P']\!]_\sigma & [\![T b]\!]_\sigma = 1, \end{cases} \end{aligned}$$

  while

$$\begin{aligned} \left([\![\texttt{if}\, b\ \texttt{then}\, P\ \texttt{else}\, P']\!]\sigma\right) x_i &= \left(\mathrm{cond}([\![b]\!], [\![P]\!], [\![P']\!])\sigma\right) x_i \\ &= \begin{cases} ([\![P]\!]\sigma) x_i & [\![b]\!]\sigma = \mathsf{tt} \\ ([\![P']\!]\sigma) x_i & [\![b]\!]\sigma = \mathsf{ff}. \end{cases} \end{aligned}$$

  We note that the two case distinctions match via the property of our translation, and that the corresponding expressions are equal by the induction hypothesis.

- For composition of programs with statements the translation becomes more complicated. The interpretation of the While program in question is easily calculated as

$$([\![P\,;S]\!]\sigma)\, x_i = ([\![S]\!]([\![P]\!]\sigma))\, x_i.$$

$$\begin{aligned} &[\![T^D_{x_i}(P\,;S)]\!]_\sigma \\ &\quad = [\![\mathsf{ifz}\ (T^D_{x_i} P)\,[(\lambda x_1 \cdots x_n : \mathsf{nat}.\ T^D_{x_i} S)(T^D_{x_1} P) \cdots (T^D_{x_n} P)]^2]\!]_\sigma \\ &\quad = \begin{cases} \bot & [\![T^D_{x_i} P]\!]_\sigma = \bot \\ [\![(\lambda x_1 \cdots x_n : \mathsf{nat}.\ T^D_{x_i} S)(T^D_{x_1} P) \cdots (T^D_{x_n} P)]\!]_\sigma & \text{else} \end{cases} \end{aligned}$$

  By the induction hypothesis $[\![T^D_{x_i} P]\!]_\sigma = \bot$ if and only if $[\![P]\!]\sigma = \bot$, and so the two denotations agree in that case since we know $[\![S]\!]$ to be strict. For the else case, we observe that

$$\begin{aligned} &[\![(\lambda x_1 . \cdots . x_n : \mathsf{nat}.\ T^D_{x_i} S)(T^D_{x_1} P) \cdots (T^D_{x_n} P)]\!]_\sigma \\ &\quad = [\![T^D_{x_i} S]\!]_{\sigma[x_1 \mapsto [\![T^D_{x_1} P]\!]_\sigma] \cdots [x_n \mapsto [\![T^D_{x_n} P]\!]_\sigma]}, \end{aligned}$$

  which by the induction hypothesis is equal to

$$\left([\![S]\!](\sigma[x_1 \mapsto ([\![P]\!]\sigma) x_1] \cdots [x_n \mapsto ([\![P]\!]\sigma) x_n])\right) x_i,$$

  and since by Proposition 1.4 we have that

$$\sigma[x_1 \mapsto ([\![P]\!]\sigma) x_1] \cdots [x_n \mapsto ([\![P]\!]\sigma) x_n] = [\![P]\!]\sigma$$

  we have also shown this case.

That leaves us with understanding what happens when we have a loop in the program. We begin by giving the translation of a loop. Here we use ρ^n to denote the type that expects n many inputs of type nat to produce an output of type nat. The translation of

```
while b do P
```

is

$$\Big(\mathsf{rec}\ (\lambda\phi\colon\rho^n.\ \lambda x_1\cdots x_n\colon\mathsf{nat}.\ \mathsf{ifz}\,(T\,b)\ \Big[\mathsf{ifz}\ (T^D_{x_i}\,P)\,[\phi(T^D_{x_1}\,P)\cdots(T^D_{x_n}\,P)]^2\Big]\ \big[x_i\big]\big)\Big)\ x_1\cdots x_n.$$

The interpretation of the resulting PCF term is also via a fixed point construction, but for the function G whose type is $[\![\rho^n]\!] \longrightarrow [\![\rho^n]\!]$. In other words, G is an endofunction on

$$\underbrace{\mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow (\cdots \mathbb{N}_\perp}_{n\text{ many}} \Rightarrow \mathbb{N}_\perp)\cdots).$$

We discuss the interpretation of this PCF term, and its connection with the interpretation of the original program, in Section 3. To obtain a working understanding we look at examples first.

2.3 Examples

We revisit the examples from Section 1.6 to improve our understanding of how the translation works, and how the interpretation of a translated program might relate to the interpretation of the original.

Based on the material presented in the previous section our only concern is understanding the translation of loops, and we concentrate on those.

Instead of looking at the full translated term we use some simplifications that have the same interpretation. For example, if we know that the body of the loop terminates, we do not use the second conditional in the given translation, and we also take shortcuts with arithmetic and boolean expressions.

Non-terminating loop

We again look at a trivial loop in the form of the following program.

```
while true do skip
```

This program mentions no variables but we may nonetheless consider the PCF term T^D_x, where $D = \{x\}$. Its translation has the same interpretation as the PCF term

$$(\mathsf{rec}\,(\lambda\phi\colon\mathsf{nat}\to\mathsf{nat}.\ \lambda x\colon\mathsf{nat}.\ \mathsf{ifz}\,\bar{0}[\phi x][x]))\ x.$$

The interpretation of the term inside the rec constructor is the least fixed point of the function

$$G\colon\ \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp$$

which for $g\colon\ \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp$ and $n \in \mathbb{N}$ is given by

$$\begin{aligned} Ggn &= \mathrm{ifz}\ 0\,gn\,n \\ &= gn. \end{aligned}$$

In order to compute the corresponding fixed point we have to work out what happens if we repeatedly apply G to $k_\perp$, and we can see that

$$Gk_\perp = k_\perp,$$

so $k_\perp$ is the least fixed point of G. A priori this looks like the same result as we had for $[\![P]\!]$ in Section 1.6, but we need to be careful here in that the two functions $k_\perp$ have different types in the two settings. But we may certainly verify that we have, for a state σ that $[\![P]\!]\sigma = \perp \in \mathsf{St}_\perp$ and so $([\![P]\!]\sigma)x = \perp$, as well as $[\![P]\!]_\sigma = \perp \in \mathbb{N}_\perp$.

Terminating loop without effect

We next look at

```
while false do skip.
```

Its translation for the variable x has the same interpretation as the PCF term

$$(\mathsf{rec}\,(\lambda\phi\colon \mathsf{nat} \to \mathsf{nat}.\ \lambda x\colon \mathsf{nat}.\ \mathsf{ifz}\,(\bar{\mathsf{s}}\bar{\mathsf{0}})[\phi x][x]))\ x.$$

The interpretation of the term inside the rec constructor is the least fixed point of the function

$$G\colon \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp$$

which for $g\colon \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp$ and $n \in \mathbb{N}$ is given by

$$\begin{aligned} Ggn &= \mathrm{ifz}\ 1\ gn\,n \\ &= n. \end{aligned}$$

We may see that $Gk_\perp = \mathrm{id}_{\mathbb{N}_\perp}$, and that the identity function on $\mathbb{N}_\perp$ is the least fixed point of G. The interpretation of the full term, for a state σ, is the application of this function to the number σx, which gives σx.

Hence we do have that

$$([\![P]\!]\sigma)x = (\mathrm{id}_{\mathsf{St}_\perp}\,\sigma)x = \sigma x = [\![T^D_{x_i}\,P]\!]_\sigma$$

in this case.

Simple terminating loop with limited effect

Our third example is once again the program

```
while ~(x=1) do x:=1.
```

Its translation has the same interpretation as the PCF term

$$\left(\mathsf{rec}\,(\lambda\phi\colon \mathsf{nat} \to \mathsf{nat}.\ \lambda x\colon \mathsf{nat}.\ \mathsf{ifz}\,(t_\sim t_= x(\bar{\mathsf{s}}\bar{\mathsf{0}}))[\phi\bar{\mathsf{s}}\bar{\mathsf{0}}][x])\right) x$$

where $t_\sim$ and $t_=$ are terms that match the expected behaviour of the given boolean expression. The interpretation of the term inside the rec constructor is the least fixed point of the function

$$G\colon \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp$$

which for $g\colon \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp$ and $n \in \mathbb{N}$ is given by

$$Ggn = \mathrm{ifz}\,(n \neq 1)\,g1\,n$$

$$= \begin{cases} \bot & n = \bot \\ g1 & n \in \mathbb{N},\, n \neq 1 \\ 1 & \text{else,} \end{cases}$$

where we assume an appropriate interpretation of $n \neq 1$ as first input to ifz. We may now calculate that

$$Gk_\bot n = \begin{cases} \bot & n \neq 1 \\ 1 & \text{else,} \end{cases}$$

and if we apply G again we obtain

$$G^2 k_\bot n = \begin{cases} Gk_\bot 1 & n \neq 1 \\ 1 & \text{else} \end{cases} = \begin{cases} 1 & n \neq 1 \\ 1 & \text{else} \end{cases} = 1.$$

This is the least fixed point of G, and applying it to σx is the interpretation of the full PCF term from above. Hence once again we may confirm that we have, for a state σ, that

$$(\llbracket P \rrbracket \sigma)x = (\sigma[x \mapsto 1])x = 1 = (\mathsf{fix}\, G)(\sigma x) = \llbracket T^D_{x_i}\, P \rrbracket_\sigma$$

in this case.

Loop with varying termination behaviour

We finally once again look at a situation where the loop sometimes terminates, and sometimes doesn't, depending on the state we start in.

```
while ~(x=1) do x:=y
```

Note that this program uses two variables, which causes some changes to our translation, and correspondingly to the denotation of the resulting PCF term.

The translation of the given program for the output variable x has the same interpretation as the PCF term

$$\big(\mathsf{rec}\,(\lambda\phi : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\, \lambda x : \mathsf{nat}.\, \lambda y : \mathsf{nat}.\, \mathsf{ifz}\,(t_\sim t_= x(\bar{\mathsf{s}}\bar{\mathsf{0}}))[\phi y y][x])\big)\; x\; y$$

The interpretation of the term inside the rec constructor is the least fixed point of the function

$$G : \mathbb{N}_\bot \Rightarrow (\mathbb{N}_\bot \Rightarrow \mathbb{N}_\bot) \longrightarrow \mathbb{N}_\bot \Rightarrow (\mathbb{N}_\bot \Rightarrow \mathbb{N}_\bot)$$

which for $g : \mathbb{N}_\bot \longrightarrow \mathbb{N}_\bot$ and $m, n \in \mathbb{N}$ is given by

$$\begin{aligned} Ggmn &= \mathrm{ifz}\,(m \neq 1)\,(gnn)\,m \\ &= \begin{cases} \bot & m = \bot \\ gnn & m \in \mathbb{N} \text{ and } m \neq 1 \\ m & \text{else.} \end{cases} \end{aligned}$$

We now require a function which we again call $k_\bot$ with the behaviour that $kmn = \bot$ for all inputs m and n. We calculate

$$Gk_\bot mn = \begin{cases} \bot & m \neq 1 \\ m & \text{else} \end{cases} = \begin{cases} \bot & m \neq 1 \\ 1 & \text{else,} \end{cases}$$

and if we apply G again we obtain

$$G^2 k_\perp mn = \begin{cases} \perp & m = \perp \\ G k_\perp nn & m \in \mathbb{N} \text{ and } m \neq 1 \\ m & \text{else} \end{cases} = \begin{cases} \perp & m = \perp \\ \perp & m \in \mathbb{N} \text{ and } m \neq 1 \\ 1 & \end{cases} = \begin{cases} \perp & m \neq 1 \text{ and } n \neq 1 \\ 1 & \text{else} \end{cases}$$

This is the least fixed point of G which is applied to σx to give the interpretation of the PCF term given. Hence once again we may confirm that we have, for a state σ, if $x = 1$ or $y = 1$ we have

$$(\llbracket P \rrbracket \sigma)x = (\sigma[x \mapsto 1])x = 1 = (\mathsf{fix}\, G)(\sigma x) = \llbracket T^D_{x_i}\, P \rrbracket_\sigma$$

while otherwise $(\llbracket P \rrbracket \sigma)x = \perp = \llbracket T^D_{x_i}\, P \rrbracket_\sigma$.

If instead we calculate the translation for the output variable y we obtain a PCF term that has the same interpretation as

$$\left(\mathsf{rec}\,(\lambda\phi : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{ifz}\,(t_{\sim} t_{=} x(\overline{\mathsf{s}}\overline{0}))[\phi y y][y])\right)\, x\, y$$

and the corresponding function G is given by

$$\begin{aligned} G g m n &= \mathrm{ifz}\,(m \neq 1)\,(g n n)\, n \\ &= \begin{cases} \perp & m = \perp \\ g n n & m \in \mathbb{N} \text{ and } m \neq 1 \\ n & \text{else.} \end{cases} \end{aligned}$$

Now

$$G k_\perp mn = \begin{cases} \perp & m \neq 1 \\ n & \text{else} \end{cases}$$

while

$$G^2 k_\perp mn = \begin{cases} \perp & m = \perp \\ G k_\perp nn & m \neq 1 \\ n & \text{else} \end{cases} = \begin{cases} \perp & m \neq 1 \text{ and } n \neq 1 \\ n & m \neq 1 \text{ and } n = 1 \\ n & \text{else} \end{cases} = \begin{cases} \perp & m \neq 1 \text{ and } n \neq 1 \\ n & \text{else} \end{cases}$$

which gives the required fixed point. This is applied to σy for the interpretation of $T^D_y\, P$, and this confirms that we have, for a state σ that if $\sigma x = 1$ or $\sigma y = 1$ then

$$(\llbracket P \rrbracket \sigma)y = \sigma y = (\mathsf{fix}\, G)(\sigma y) = \llbracket T^D_y\, P \rrbracket_\sigma$$

and otherwise both expressions evaluate to $\perp \in \mathbb{N}_\perp$.

These examples illustrate a few interesting points:

- The calculation of the least fixed point is closely related to that of the least fixed point that appears in the interpretation of the given While program.
- In order to calculate the required fixed point we make no use of the given state (which appears in the valuation we require to interpret the translation)—the calculation works for all possible states, and it is only when we evaluate the fixed point that we feed the values of the relevant variable for a fixed state to the term.

These operations raise hope that there may be a simple connection between the two.

3 Connecting the two fixed point constructions

We provide a connection between the two fixed point constructions. This is motivated by wanting to complete the proof by induction of statement (∗), which says that if Q is a While program, σ is a state and x a variable, we have

$$(\llbracket Q \rrbracket \sigma)x = \llbracket T^D_{x_i} Q \rrbracket_\sigma.$$

3.1 The interpretation of loops

The missing step case is that where the program Q is of the form

while b **do** P**.**

The interpretation $\llbracket Q \rrbracket$ of the While program Q is the least fixed point of the function

$$\begin{aligned} F_Q : \mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp &\longrightarrow \mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp \\ f &\longmapsto \mathrm{cond}(\llbracket b \rrbracket, f \circ \llbracket P \rrbracket, \mathrm{id}_{\mathsf{St}_\perp}) \end{aligned}$$

In order to understand this assignment we look at its effect on inputs $f \in \mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$ and $\tau \in \mathsf{St}_\perp$, which is given by

$$F_Q f \tau = \begin{cases} \perp & \tau = \perp \\ f(\llbracket P \rrbracket \tau) & \llbracket b \rrbracket \tau = \mathsf{tt} \\ \tau & \llbracket b \rrbracket \tau = \mathsf{ff}. \end{cases}$$

For comparison with the function that interprets the translation of Q we find it useful to make reorder the case distinctions and add another, using the fact that f is strict:

$$F_Q f \tau = \begin{cases} \perp & \tau = \perp \\ \tau & \llbracket b \rrbracket \tau = \mathsf{ff} \\ \perp & \llbracket b \rrbracket \tau = \mathsf{tt} \text{ and } \llbracket P \rrbracket \tau = \perp \\ f(\llbracket P \rrbracket \tau) & \llbracket b \rrbracket \tau = \mathsf{tt} \text{ and } \llbracket P \rrbracket \tau \neq \perp \end{cases}$$

For this definition we have

$$\llbracket Q \rrbracket = \mathsf{fix}\, F_Q.$$

The interpretation of the translation of Q is relative to a state σ, and it requires us to know that $x_1, x_2, \ldots, x_n$ contain all the variables that occur in Q. We also have to choose an output variable x_i.

This interpretation also relies on a fixed point construction. Here we are interested in the function

$$G^i_Q : \underbrace{\mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow (\cdots \mathbb{N}_\perp}_{n \text{ many}} \Rightarrow \mathbb{N}_\perp) \cdots) \longrightarrow \underbrace{\mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow (\cdots \mathbb{N}_\perp}_{n \text{ many}} \Rightarrow \mathbb{N}_\perp) \cdots).$$

We sometimes find it easier to supply arguments to a function in the source of G_i as tuples, and we note that by Fact 1 we have

$$\underbrace{\mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow (\cdots \mathbb{N}_\perp}_{n \text{ many}} \Rightarrow \mathbb{N}_\perp) \cdots) \cong \mathbb{N}_\perp^n \Rightarrow \mathbb{N}_\perp,$$

and we sometimes use the corresponding alternative typing for G^i_Q.

The least fixed point of G^i_Q is the interpretation of the subterm T of $T^D_{x_i}\,Q$ to which the rec constructor applies. A straightforward proof by induction ensures that T does not contain any free variables: The only variables that may appear in a translated term are:

- When translating expressions we invoke terms that implement the various required operations, but these terms contain no free variables, and so the only potential free variables come from the base case where we translate a While variable to the corresponding PCF variable. In T these are all abstracted.
- When translating programs, we use the variables from D as well as ϕ. Again these variables are abstracted in T and so they do not occur free in T.

By the abstraction step case of the definition of the denotational semantics for PCF, the interpretation of subterms of T come with the valuation

$$[\phi \mapsto g][x_1 \mapsto m_1] \cdots [x_n \mapsto m_n],$$

where g is a function in the interpretation of the type of ϕ, and the m_i are elements in $\mathbb{N}_\perp$.

We further know where ϕ occurs as a free variable, which allows us to put g in the one place where it is required, and this means that we may write the assignment underlying G^i_Q by referring to the valuation

$$\theta = [x_1 \mapsto m_1] \cdots [x_n \mapsto m_n],$$

where we no longer need to worry about ϕ.

These are the arguments required by G^i_Q, and we may describe its underlying assignment as

$$G^i_Q\, g\, m_1 m_2 \cdots m_n = \begin{cases} \perp & [\![T\,b]\!]_\theta = \perp \\ m_i & [\![T\,b]\!]_\theta = 1 \\ \perp & [\![T\,b]\!]_\theta = 0 \text{ and } [\![T^D_{x_i}\,P]\!]_\theta = \perp \\ g\,[\![T^D_{x_1}\,P]\!]_\theta \cdots [\![T^D_{x_n}\,P]\!]_\theta & \text{else.} \end{cases}$$

It is important here to note that the expression $[\![T\,b]\!]_\theta$ *depends on the inputs* m_i *under consideration*, which means that if we apply G^i_Q repeatedly, as is required for the construction of the fixed point, *the case distinctions may (and typically will) change.* The fourth example in Section 2.3 illustrates this.

The two assignments show some similarities which we want to exploit.

We observe that G^i_Q is calculated independent from the state σ relative to which we calculate the interpretation of the translation of Q—in some sense, $\mathsf{fix}\, G^i_Q$ is calculated for all possible states. We also note that the G^i_Q are uniform in their construction, and differ in only one place depending on which output variable x_i is chosen.

To establish the property $(*)$ we need to show that, given a state σ and a variable x_i mentioned in Q, we have

$$((\mathsf{fix}\, F_Q)\sigma)x_i = (\mathsf{fix}\, G^i_Q)\,(\sigma x_1) \cdots (\sigma x_n).$$

Our strategy is to do this by proving a connection between the functions F_Q and the G^i_Q.

3.2 Laying the ground work

We methodically build a connection between F_Q and the $G^i Q$ which eventually allows us to make the desired connection between their fixed points.

The case distinctions of the two functions are controlled by the interpretations of boolean expressions, and arithmetic expressions, and we collect suitable properties of those first.

In order to tackle programs we then have to properly account for the values which we use to denote non-termination, and relate

$$\mathsf{St}_\perp \qquad \text{and} \qquad \prod_{1\le i\le n} \mathbb{N}_\perp,$$

which we may do by extending the families of functions zr and π from Section 1.2.

We then check that $\mathrm{fix}\, F_Q$ and the $\mathrm{fix}\, G^i_Q$ have the same termination behaviour, before we finally show that there is a sufficient connection between F_Q and G^i_Q to argue that their fixed points satisfy the required equality.

3.3 Expressions

The interpretation of expressions is simpler since they are given by functions from states to the natural numbers or the booleans, rather than having states as part of the output. It is easy to establish that for every arithmetic expression a, and every boolean expression b, as well as states σ and σ':

If the variables mentioned in a (or b) are included among $x_1, x_2, \ldots, x_n$ then

$$\pi_{x_1,\ldots,x_n}\sigma = \pi_{x_1,\ldots,x_n}\sigma' \qquad \text{implies} \qquad [\![a]\!]\sigma = [\![a]\!]\sigma' \ (\text{or } [\![b]\!]\sigma = [\![b]\!]\sigma').$$

This implies further that for all states σ we have

$$[\![a]\!]\sigma = [\![a]\!](\mathrm{zr}_{x_1,\ldots,x_n}(\pi_{x_1,\ldots,x_n}\sigma)),$$

and similarly for boolean expressions.

3.4 Programs

Since programs manipulate states, and in particular output a state (or $\perp$), the interaction between their interpretation and the functions F_Q and G^i_Q from above is less straightforward.

We have to extend our definition of the family of functions π to cope with the potential input $\perp$ and so create functions

$$\mathsf{St}_\perp \longrightarrow \textstyle\prod_{1\le i\le n} \mathbb{N}_\perp.$$

The obvious way of doing so is to define an assignment

$$\tau \longmapsto \begin{cases} (\perp)_{x_1,\ldots,x_n} & \tau = \perp \\ \pi_{x_1,\ldots,x_n}\tau & \text{else.} \end{cases}$$

In order to avoid a surfeit of subscripts, we take the liberty of re-using the name $\pi_{x_1,\ldots,x_n}$ for this function.

We would also like to reconsider the typing of our families of function zr to be

$$\textstyle\prod_{1\le i\le n} \mathbb{N}_\perp \longrightarrow \mathsf{St}_\perp$$

which we do by the assignment

$$(m_1 \cdots m_n) \longmapsto \begin{cases} \bot & \exists i.\, m_i = \bot \\ \mathrm{zr}_{x_1,\ldots,x_n}(m_1 \cdots m_n) & \text{else.} \end{cases}$$

Again we abuse notation by referring to this function as $\mathrm{zr}_{x_1,\ldots,x_n}$. In the text below whenever we use these names we mean the functions typed as above.

Assume we have a programme P whose variables are among x_1, x_2,..., x_n.

The equalities we give following Proposition 1.4 for the functions π and zr extend to these redefined versions:

If σ and σ' are two states that agree for these variables then we may express this as saying that

$$\pi_{x_1,\ldots,x_n}\sigma = \pi_{x_1,\ldots,x_n}\sigma',$$

and in this situation we have

$$\pi_{x_1,\ldots,x_n}([\![P]\!]\sigma) = \pi_{x_1,\ldots,x_n}([\![P]\!]\sigma').$$

We further have that Property (†) from above still holds for these functions, that is

$$\mathrm{zr}_{x_1,\ldots,x_n} \circ \pi_{x_1,\ldots,x_n} \circ [\![P]\!] \circ \mathrm{zr}_{x_1,\ldots,x_n} = [\![P]\!] \circ \mathrm{zr}_{x_1,\ldots,x_n} \qquad (\dagger).$$

We may now reformulate our desired property as

$$((\mathsf{fix}\, F_Q)\sigma)x_i = (\pi_{x_i} \circ (\mathsf{fix}\, F_Q))\sigma = (\mathsf{fix}\, G_Q^i)\pi_{x_1,\ldots,x_n}\sigma = (\mathsf{fix}\, G_Q^i)(\sigma x_1)\cdots(\sigma x_n),$$

or

$$\pi_{x_i} \circ (\mathsf{fix}\, F_Q) = (\mathsf{fix}\, G_Q^i) \circ \pi_{x_1,\ldots,x_n}. \qquad (**)$$

3.5 Termination

We may see that the definitions of F_Q and the G_Q^i make case distinctions that depend on the termination of P or, to be more precise, on $[\![P]\!]\sigma = \bot$ or $[\![T^D_{x_i} P]\!]_\theta = \bot$, and we are only concerned about the situation where the values that appear in θ are natural numbers rather than the bottom element of $\mathbb{N}_\bot$.

We aim to show by induction that the two match, provided σ and θ correspond to each other. By this we mean that for

$$\theta = [x_1 \mapsto m_1]\cdots[x_n \mapsto m_n]$$

we have

$$\pi_{x_1,\ldots,x_n}\sigma = (m_1, \ldots, m_n),$$

that is, the values assigned to the variables that occur in P coincide with those provided by the valuation θ.

It is our aim to establish that

$$[\![P]\!]\sigma = \bot \qquad \text{if and only if} \qquad [\![T^D_{x_i} P]\!]_\theta = \bot,$$

where

$$\theta = [x_1 \mapsto \sigma x_1]\cdots[x_n \mapsto \sigma x_n].$$

In Section 1.5 we argue that the only case where non-termination may arise for a While program is that of a loop. It is further established in Section 2.2 that for the other cases, the

interpretation of a program agrees with that of its translation for a state σ. By Proposition 1.4 the interpretation of a While program only depends on the values for the variables that occur in it. This means our desired property holds for these cases, and the only potential issue is that of the loop.

We begin by checking the fixed point of the function F_Q that arises in the interpretation of the program Q

while b **do** P.

For $k_\perp$ the constant $\perp$ function from $\mathsf{St}_\perp \Rightarrow_{\mathsf{s}} \mathsf{St}_\perp$ to itself we have

$$\mathsf{fix}\, F_Q = \bigvee_{j\in\mathbb{N}} F_Q^j k_\perp.$$

We know by Proposition 4.4 from [RS22] that

$$\bigvee_{j\in\mathbb{N}} F_Q^j = \bigvee_{j\in\mathbb{N}} F_Q^{j+1},$$

and so we may calculate the fixed point as

$$\begin{aligned}
\left(\bigvee_{j\in\mathbb{N}} F_Q^{j+2}\right) k_\perp \tau &= \bigvee_{j\in\mathbb{N}} (F_Q^{j+2} k_\perp)\tau \\
&= \bigvee_{j\in\mathbb{N}} \begin{cases} \perp & \tau = \perp \\ \tau & [\![b]\!]\tau = \mathsf{ff} \\ (F_Q^{j+1}k_\perp)([\![P]\!]\tau) & [\![b]\!]\tau = \mathsf{tt}. \end{cases} \\
&= \bigvee_{j\in\mathbb{N}} \begin{cases} \perp & \tau = \perp \\ \tau & [\![b]\!]\tau = \mathsf{ff} \\ \perp & [\![b]\!]\tau = \mathsf{tt} \text{ and } [\![P]\!]\tau = \perp \\ [\![P]\!]\tau & [\![b]\!]\tau = \mathsf{tt} \text{ and } [\![b]\!]([\![P]\!]\tau) = \mathsf{ff} \\ (F_Q^j k_\perp)([\![P]\!]^2\tau) & \text{else.} \end{cases} \\
&= \bigvee_{j\in\mathbb{N}} \begin{cases} \perp & \tau = \perp \text{ or } ([\![b]\!]\tau = \mathsf{tt} \text{ and } [\![P]\!]\tau = \perp) \\ \tau & [\![b]\!]\tau = \mathsf{ff} \\ [\![P]\!]\tau & [\![b]\!]\tau = \mathsf{tt} \text{ and } [\![b]\!]([\![P]\!]\tau) = \mathsf{ff} \\ (F_Q^j k_\perp)([\![P]\!]^2\tau) & \text{else} \end{cases} \\
&= \bigvee_{j\in\mathbb{N}} \begin{cases} \perp & \tau = \perp \text{ or } ([\![b]\!]\tau = \mathsf{tt} \text{ and } ([\![P]\!]\tau = \perp \text{ or} \\ & \qquad ([\![b]\!]([\![P]\!]\tau) = \mathsf{tt} \text{ and } [\![P]\!]^2\tau = \perp))) \\ \tau & [\![b]\!]\tau = \mathsf{ff} \\ [\![P]\!]\tau & [\![b]\!]\tau = \mathsf{tt} \text{ and } [\![b]\!]([\![P]\!]\tau) = \mathsf{ff} \\ (F_Q^j k_\perp)([\![P]\!]^2\tau) & \text{else.} \end{cases}
\end{aligned}$$

We further know from Proposition 1.2 that

$$\mathsf{fix}\, F_Q \sigma = \perp$$

if and only if it is the case that for all $j \in \mathbb{N}$ we have

$$(F_Q^j k_\perp)\sigma = \perp.$$

Inspecting the directed supremum from above we may see that it takes on the value $\bot$ for all j for an input τ if one of the following happens:

- It is the case that $\tau = \bot$ in $\mathsf{St}_\bot$ or
- $[\![b]\!]\,\tau = \mathrm{tt}$ and $[\![P]\!]\,\tau = \bot$ or
- there exists $k \in \mathbb{N}$ such that $[\![b]\!]([\![P]\!]^l \tau) = \mathrm{tt}$ and $[\![P]\!]^l \neq \bot$ for $1 \leq l < k$, and $[\![P]\!]^k = \bot$ (this is a generalization of the previous case) or
- on every iteration we are in the final case which can only happen if $[\![b]\!]([\![P]\!]^j \tau) = \mathrm{tt}$ for all $j \in \mathbb{N}$.

We similarly expand the expression obtained when calculating the least fixed point of G_Q^i. If we are in the fourth case of the following case distinction and we apply G_Q^i again, we need to account for the fact that θ is a function in the m_i which needs to be updated to

$$\theta' = [x_1 \mapsto [\![T^D_{x_1}\, P]\!]_\theta] \cdots [x_n \mapsto [\![T^D_{x_n}\, P]\!]_\theta].$$

$$\left(\bigvee_{j\in\mathbb{N}} (G_Q^i)^{j+2}\right) k_\bot m_1 \cdots m_n$$

$$= \bigvee_{j\in\mathbb{N}} ((G_Q^i)^{j+2} k_\bot) m_1 \cdots m_n$$

$$= \bigvee_{j\in\mathbb{N}} \begin{cases} \bot & [\![Tb]\!]_\theta = \bot \\ m_i & [\![Tb]\!]_\theta = 1 \\ \bot & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta = \bot \\ ((G_Q^i)^{j+1} k_\bot)([\![T^D_{x_1}\, P]\!]_\theta) \cdots ([\![T^D_{x_n}\, P]\!]_\theta) & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta \neq \bot. \end{cases}$$

$$= \bigvee_{j\in\mathbb{N}} \begin{cases} \bot & [\![Tb]\!]_\theta = \bot \\ m_i & [\![Tb]\!]_\theta = 1 \\ \bot & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta = \bot \\ \bot & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta \neq \bot,\ [\![Tb]\!]_{\theta'} = \bot \\ [\![T^D_{x_i}\, P]\!]_\theta & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta \neq \bot,\ [\![Tb]\!]_{\theta'} = 1 \\ \bot & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta \neq \bot,\ [\![Tb]\!]_{\theta'} = 0,\ [\![T^D_{x_i}\, P]\!]_{\theta'} = \bot \\ ((G_Q^i)^j k_\bot)([\![T^D_{x_1}\, P]\!]_{\theta'}) \cdots ([\![T^D_{x_n}\, P]\!]_{\theta'}) & \text{else} \end{cases}$$

$$= \bigvee_{j\in\mathbb{N}} \begin{cases} \bot & [\![Tb]\!]_\theta = \bot \text{ or } ([\![Tb]\!]_\theta = 0 \text{ and } ([\![T^D_{x_i}\, P]\!]_\theta = \bot \text{ or} \\ & \quad [\![Tb]\!]_{\theta'} = \bot \text{ or } ([\![Tb]\!]_{\theta'} = 0 \text{ and } [\![T^D_{x_i}\, P]\!]_{\theta'} = \bot))) \\ m_i & [\![Tb]\!]_\theta = 1 \\ [\![T^D_{x_i}\, P]\!]_\theta & [\![Tb]\!]_\theta = 0,\ [\![T^D_{x_i}\, P]\!]_\theta \neq \bot,\ [\![Tb]\!]_{\theta'} = 1 \\ ((G_Q^i)^j k_\bot)([\![T^D_{x_1}\, P]\!]_{\theta'}) \cdots ([\![T^D_{x_n}\, P]\!]_{\theta'}) & \text{else.} \end{cases}$$

Note that by the induction hypothesis we know that

$$[\![T^D_{x_k}\, P]\!]_{\theta'} = \left([\![P]\!]\ \mathrm{zr}_{x_1,\ldots,x_n}(([\![T^D_{x_1}\, P]\!]_\theta) \cdots ([\![T^D_{x_n}\, P]\!]_\theta))\right) x_k = ([\![P]\!]^2 \theta) x_k$$

for all $1 \leq k \leq n$. This means that we may describe θ' as

$$\pi_{x_1,\ldots,x_n}([\![P]\!]\ \mathrm{zr}_{x_1,\ldots,x_n}(m_1,\ldots,m_n)),$$

and more generally, that on the $j+1$th iteration of G we have the valuation given by

$$\theta^j = \pi_{x_1,\ldots,x_n}([\![P]\!]^j\ \mathrm{zr}_{x_1,\ldots,x_n}(m_1,\ldots,m_n)).$$

It is further useful to realize that if $[\![Tb]\!]_\theta \neq \bot$ and $[\![T^D_{x_i}\ P]\!]_\theta \neq \bot$ then we also know that $[\![Tb]\!]_{\theta^j} \neq \bot$, which is why there are no conditions for the latter.

We may similarly argue that for $(\mathrm{fix}\, G^i_Q)\, m_1 \cdots m_n = \bot$ to hold it must be the case that this is true for all iterations of $(G^i_Q)^j$, where $j \in \mathbb{N}$, for the inputs $g = k_\bot$ and $m_1, \ldots, m_n$ from $\mathbb{N}$. This happens in the following cases:

- If $[\![Tb]\!]_\theta = \bot$, but this can only happen if one of the m_i is equal to $\bot$, and we are only interested in the case where all the m_i are in $\mathbb{N}$.
- If $[\![T^D_{x_i}\ b]\!]_\theta = 0$ and $[\![T^D_{x_i}\ P]\!]_\theta = \bot$, and this matches the second case for F_Q from above using the induction hypothesis and the properties of our translation for regular expressions.
- There exists $l \in \mathbb{N}$ such that $[\![Tb]\!]_\theta = 0$ and for all $k < l$ it is the case that $[\![Tb]\!]_{\theta^{k+1}} = 0$, $[\![T^D_{x_i}\ P]\!]_{\theta^k} \neq \bot$ and $[\![T^D_{x_i}\ P]\!]_{\theta^l} = \bot$, which matches the third case for F_Q form for the same reason.
- If on every iteration we are in the final case, which means that for all $j \in \mathbb{N}$ we have $[\![b]\!]_{\theta^j} = 0$ for all $j \in \mathbb{N}$. This matches the fourth case give for F_Q from above.

Hence we have for all While programs P that

$$[\![P]\!]\ \mathrm{zr}_{x_1,\ldots,x_n}(m_1 \cdots m_n) = \bot \qquad \text{if and only if} \qquad [\![T^D_{x_i}\ P]\!]_\theta = \bot,$$

and we use that property below to show that the two fixed points are related as we expect them to be.

3.6 Connecting F_Q with the G^i_Q

We can see that based on Fact 1 we may connect the two functions of interest in the following way:

$$\begin{array}{ccc} \mathsf{St}_\bot \Rightarrow_{\mathsf{s}} \mathsf{St}_\bot & \xrightarrow{F_Q} & \mathsf{St}_\bot \Rightarrow_{\mathsf{s}} \mathsf{St}_\bot \\ {\scriptstyle \mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}}\downarrow & & \downarrow{\scriptstyle \mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}} \\ \mathbb{N}^n_\bot \Rightarrow \mathbb{N}_\bot & \xrightarrow[G^i_Q]{} & \mathbb{N}^n_\bot \Rightarrow \mathbb{N}_\bot \end{array}$$

The aim is to show that we have the required properties to invoke Proposition 4.2 which forges a connection between $\mathrm{fix}\, F_Q$ and $\mathrm{fix}\, G^i_Q$ by telling us that that

$$\mathrm{fix}\, G^i_Q \pi_{\vec{x}}\sigma = \pi_{x_i}(\mathrm{fix}\, F_Q(\mathrm{zr}_{\vec{x}}(\pi_{\vec{x}}\sigma))). \qquad (**)$$

We may think of $\pi_{\vec{x}}\sigma$ as feeding the values from the valuation given by σ to $\mathrm{fix}\, G^i_Q$. Using this property we may further argue that we have for all states σ that

$$[\![T^D_{x_i}\ Q]\!]_\sigma = (\mathrm{fix}\, G^i_Q)\sigma x_1 \cdots \sigma x_n \qquad \text{def } [\![\,]\!]_\sigma$$

$$
\begin{aligned}
&= (\mathsf{fix}\, G_Q^i)\pi_{\vec{x}}\sigma && \text{def } \pi_{\vec{x}} \\
&= \pi_{x_i}\left((\mathsf{fix}\, F_Q)(\mathrm{zr}_{\vec{x}}\, \pi_{\vec{x}}\sigma)\right) && \text{Property } (**) \\
&= \left(\llbracket Q \rrbracket (\mathrm{zr}_{\vec{x}}\, \pi_{\vec{x}}\sigma)\right) x_i && \text{def } \llbracket\, \rrbracket \\
&= (\llbracket Q \rrbracket \sigma) x_i && \text{Proposition 1.4,}
\end{aligned}
$$

which establishes that the translation does preserve the meaning of a program, see also Theorem 3.1.

It remains therefore to establish property $(**)$, and in order to do so we merely have to show that the conditions from Proposition 4.2 are satisfied, that is we have to show that $\pi_{x_i} \Rightarrow \mathrm{zr}_{\vec{x}}$ is strict and that the diagram from above commutes, that is

$$
G_Q^i \circ (\mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}) = (\mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}) \circ F_Q,
$$

but by Proposition 4.1 it is sufficient to do so for inputs other than $\bot$.

For the former we note that the bottom element of $\mathsf{St}_\bot \Rightarrow_\mathsf{s} \mathsf{St}_\bot$ is the function $k_\bot$ that maps every element of $\mathsf{St}_\bot$ to $\bot \in \mathsf{St}_\bot$. We have that

$$
\pi_{x_i} \circ k_\bot \circ \mathrm{zr}_{\vec{x}}
$$

is a function which, given inputs $(m_1 \cdots m_n) \in \mathbb{N}_\bot^n$, returns

$$
\begin{aligned}
\pi_{x_i}(k_\bot(\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n))) &= \pi_{x_i}\bot && \text{def } k_\bot \\
&= \bot && \text{def } \pi_{x_i}.
\end{aligned}
$$

For the latter let f be a strict function from $\mathsf{St}_\bot$ to itself, and let $(m_1 \cdots m_n)$ be an element of $\mathbb{N}^n$. For G_Q^i we obtain

$$
\begin{aligned}
&\left((G_Q^i \circ (\mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}))f\right)(m_1 \cdots m_n) \\
&\quad = \left(G_Q^i(\pi_{x_i} \circ f \circ \mathrm{zr}_{\vec{x}})\right)(m_1 \cdots m_n) \\
&\quad = \begin{cases}
m_i & \llbracket Tb \rrbracket_{\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)} = 1 \\
\bot & \llbracket Tb \rrbracket_{\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)} = 0,\ \llbracket T_{x_i}^D\, P \rrbracket_{\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)} = \bot \\
\pi_{x_i}(f\ \mathrm{zr}_{\vec{x}}(\llbracket T_{x_k}^D\, P \rrbracket_{\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)})_{1 \le k \le n}) & \text{else}
\end{cases}
\end{aligned}
$$

while for F_Q we have

$$
\begin{aligned}
&\left(((\mathrm{zr}_{\vec{x}} \Rightarrow \pi_{x_i}) \circ F_Q)f\right)(m_1 \cdots m_n) \\
&\quad = \left(\pi_{x_i} \circ F_Q f \circ \mathrm{zr}_{\vec{x}}\right)(m_1 \cdots m_n) \\
&\quad = \begin{cases}
\pi_{x_i}(\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)) = m_i & \llbracket b \rrbracket\ \mathrm{zr}_{\vec{x}}(m_1 \cdots m_n) = \mathtt{ff} \\
\bot & \llbracket b \rrbracket\ \mathrm{zr}_{\vec{x}}(m_1 \cdots m_n) = \mathtt{tt},\ \llbracket P \rrbracket\ \mathrm{zr}_{\vec{x}}(m_1 \cdots m_n) = \bot \\
\pi_{x_i}(f(\llbracket P \rrbracket(\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)))) & \text{else.}
\end{cases}
\end{aligned}
$$

We observe that the case distinctions match: for the boolean expressions this is due to the fact that we have that $\llbracket b \rrbracket \sigma = \llbracket Tb \rrbracket_\sigma$ for every state σ, and in Section 3.5 we establish that for every state σ we have $\llbracket P \rrbracket = \bot$ if and only if $\llbracket T_{x_i}^D\, P \rrbracket_\sigma = \bot$.

Hence it remains to establish that the final cases agree. The crucial information here comes from the induction hypothesis which tells us that the value of the state $[\![P]\!]\ \mathrm{zr}_{\vec{x}}(m_1 \cdots m_n))$ at a variable x_k is equal to $[\![T^D_{x_k}\ P]\!]_{\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n)}$, which means that we are comparing

$$[\![P]\!](\mathrm{zr}_{\vec{x}})(m_1 \cdots m_n)$$

with

$$\mathrm{zr}_{\vec{x}}(\pi_{\vec{x}}([\![P]\!](\mathrm{zr}_{\vec{x}}(m_1 \cdots m_n))))$$

as inputs to $\pi_{x_i} \circ f$. But we know as a consequence of Proposition 1.4 that

$$\mathrm{zr}_{\vec{x}} \circ \pi_{\vec{x}} \circ [\![P]\!] \circ \mathrm{zr}_{\vec{x}} = [\![P]\!](\mathrm{zr}_{\vec{x}},$$

see property (†) from above.

With the material from Section 2.2 we have now established the following theorem from [Sir22]:

Theorem 3.1

For every While program P whose variables are among $x_1, x_2, \ldots, x_n$ it is the case that for every state σ we have

$$\pi_{x_i}[\![P]\!]\sigma = [\![T^D_{x_i}\ P]\!]_\sigma.$$

This tells us that running the given program has the same effect as evaluating the PCF term arising from the translation for the output variable x_i.

4 Mathematical background material

Above we have used some facts about dcpos that allow us to easily move between various isomorphic ones. It further creates a connection between functions that take arguments in a product space to produce a particular outcome, versus functions that expects such arguments one at a time. We have used this idea quite freely in our account.

Fact 1

We have the following for dcpos D, D', E and E' and Scott-continuous functions

$$f : D' \longrightarrow D \qquad \text{and} \qquad g : E \longrightarrow E'.$$

(i) The dcpos $(D \times D') \Rightarrow E$ and $D \Rightarrow (D' \Rightarrow E)$ are isomorphic.

(ii) There is a Scott-continuous function

$$\begin{aligned} f \Rightarrow g : D \Rightarrow E &\longrightarrow D' \Rightarrow E' \\ h &\longmapsto g \circ h \circ f. \end{aligned}$$

(iii) If we have further Scott-continuous functions between dcpos

$$f' : D'' \longrightarrow D' \qquad \text{and} \qquad g' : E' \longrightarrow E''$$

then

$$(f' \Rightarrow g') \circ (f \Rightarrow g) = (f \circ f') \Rightarrow (g' \circ g).$$

Proposition 4.1

Let G and G' be Scott-continuous endofunctions on

$$\underbrace{\mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow (\cdots \mathbb{N}_\perp}_{n \text{ many}} \Rightarrow \mathbb{N}_\perp) \cdots).$$

If for all $m_1, m_2, \ldots m_n$ in $\mathbb{N}$ and all g which provide a suitable first input to G it is the case that if

$$G\, g\, m_1 \cdots m_n = G'\, g\, m_1 \cdots m_n$$

then

$$(\mathrm{fix}\, G)\, m_1 \cdots m_n = (\mathrm{fix}\, G')\, m_1 \cdots m_n.$$

Proof. Let $k_\perp$ be the function that takes n inputs from $\mathbb{N}_\perp$ and always gives $\perp \in \mathbb{N}_\perp$ as the output. We may prove by a simple induction proof that for the given prerequisites we have that for all $i \in \mathbb{N}$ and g and $m_1, m_2, \ldots m_n$ as given

$$(G^i g)\, m_1 \cdots m_n = (G'^i g)\, m_1 \cdots m_n.$$

To calculate the least fixed point of G for inputs $m_1, m_2, \ldots, m_n$ from $\mathbb{N}_\perp$ we may calculate

$$\begin{aligned}
(\mathrm{fix}\, G)\, m_1 \cdots m_n &= \left(\bigvee\nolimits^{\uparrow}_{j \in \mathbb{N}} G^i k_\perp \right) m_1 \cdots m_n && \text{constr of fix} \\
&= \bigvee\nolimits^{\uparrow}_{j \in \mathbb{N}} (G^i k_\perp)\, m_1 \cdots m_n && \text{def } \textstyle\bigvee^{\uparrow} \\
&= \bigvee\nolimits^{\uparrow}_{j \in \mathbb{N}} (G'^i k_\perp)\, m_1 \cdots m_n && \text{above} \\
&= (\mathrm{fix}\, G')\, m_1 \cdots m_n.
\end{aligned}$$

The following result appears as early as Plotkin's Pisa notes [Plo83]:

Proposition 4.2

Assume we have Scott-continuous functions between dcpos such that f is strict and such that the following diagram commutes

$$\begin{array}{ccc}
D & \xrightarrow{F} & D \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\
E & \xrightarrow[G]{} & E,
\end{array}$$

that is $G \circ f = f \circ F$. Then

$$\mathrm{fix}\, G = f(\mathrm{fix}\, F).$$

Proof. We note that the given condition implies that for all $j \in \mathbb{N}$ we have

$$f \circ F^j = G^j \circ f \ :$$

The base case follows from $f = f$, and in the step case we may argue (using associativity

of composition) that

$$\begin{aligned} f \circ F^{j+1} &= G^j \circ f \circ F && \text{induction hypothesis} \\ &= G^{j+1} \circ f && f \circ F = G \circ f. \end{aligned}$$

We may calculate

$$\begin{aligned} f(\operatorname{fix} F) &= f\left(\bigvee_{j \in \mathbb{N}}^{\uparrow} F^j \bot\right) && \text{def fix} \\ &= \bigvee_{j \in \mathbb{N}}^{\uparrow} f(F^j \bot) && f \text{ Scott-continuous} \\ &= \bigvee_{j \in \mathbb{N}}^{\uparrow} G^j(f \bot) && \text{property from above} \\ &= \bigvee_{j \in \mathbb{N}}^{\uparrow} G^j \bot && f \text{ strict} \\ &= \operatorname{fix} G && \text{def fix}. \end{aligned}$$

References

[NN07] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, March 2007. URL: http://dx.doi.org/10.1007/978-1-84628-692-6.

[Plo83] Gordon Plotkin. Domains (Pisa Notes). available from teh authors website, 1983. URL: https://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps.

[RS22] Joe Razavi and Andrea Schalk. Giving Meaning to Programs. Lecture notes, University of Manchester, 2022.

[Sir22] Vlad Sirbu. Studying the While Programming Language. Third year project report (draft), University of Manchester, 2022.

[Str06] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. World Scientific Publishing Co, USA, 2006. URL: https://www.semanticscholar.org/paper/Domain-theoretic-foundations-of-functional-Streicher/730a2c4fe57e22dfcf62179795d877708c176280.