More fixpoints, denotationally!

JOACHIM BREITNER, unaffiliated, Germany

This document contains unpublished and work-in-progress thoughts about a denotational description of the language extension proposed in More fixpoints! (Functional Pearl) from ICFP 2023.

You are looking at the version from July 9, 2023.

1 INTRODUCTION

In the "More Fixpoints!" functional pearl, I extend a lazy pure language (Haskell) with the ability to define recursive equations involving Sets and/or Booleans, and still produce a result. Since the semantics of pure functional programming, and the semantics of solving such equations, can both be elegantly expressed using fixpoints on partial orders, it seems prudent to search for a denotational semantics that can describe this combination. In this document I note down some some experiments in that direction.

Here is an executive summary of this text:

- We use small simply typed example language extended with recursively definable booleans (Section 2.1), and give it a *call-by-name* denotational semantics (Section 2.3). Functions are no longer continuous there, but still montone, and because the denotation is a fixed point, equataional reasoning (Section 2.4) works well.
- There are reasons why simpler approaches (simpler domains, untyped semantics) do not work (Section 2.6)
- The call-by-name semantics assigns non-bottom to programs that do not actually terminate. We can fix that using a denotational semantics in the style of "Call-by-Need Is Clairvoyant Call-by-Value" to only allow knot-tied recursion (Section 3). This semantics seems to appropriately describe the behavior of our programs. It is more abstract than an operational semantics, although we cannot deny that we would prefer even more elegance.

There is more work to be done. Questions you will not (yet) find the answer here are, among others:

- Does the call-by-need cost accounting order on D_c work as intended?
- Can we perform interesting proofs, and can we do it well?
- Can we prove the semantics correct and adequate with regard to a suitable operational semantics?
- How abstract is the semantics (i.e. how does semantic equality and contextual equivalence relate), and can we improve that?

2 DENOTATIONAL SEMANTICS

The most promising approach so far is to turn the introduction forms of our recursively defineable data types into regular constructors, and move all magic into the `get` operation.

2.1 The language

As usual, we focus on a small core language to demonstrate our ideas. For our purposes, let us used a simply typed lambda calculus with recursive bindings, extended with the conventional `Bool` data type and our `RBool`:

In this section we start with a lambda calculus in ANF with recursive let expressions

$$\tau \in \text{Typ} ::= \tau \to \tau \mid \textbf{Bool} \mid \textbf{RBool}$$

Author's address: Joachim Breitner, mail@joachim-breitner.de, unaffiliated, Germany.

$$\begin{aligned}
x &\in \mathrm{Var}\\
e \in \mathrm{Exp} &::= x \mid (\lambda x.e) \mid e\ x \mid \mathbf{let}\, x = e\, \mathbf{in}\, e\\
&\mid \mathbf{True} \mid \mathbf{False} \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e\\
&\mid \mathbf{mk} \mid \mathbf{get} \mid \mathbf{and} \mid \mathbf{or}
\end{aligned}$$

The types of the operations on **Bool** and **RBool** are

$$\begin{aligned}
\mathbf{mk} &:: \mathbf{Bool} \to \mathbf{RBool}\\
\mathbf{get} &:: \mathbf{RBool} \to \mathbf{Bool}\\
\mathbf{and}, \mathbf{or} &:: \mathbf{RBool} \to \mathbf{RBool} \to \mathbf{RBool}
\end{aligned}$$

and the typing relation $\Gamma \vdash e\colon \tau$ is standard.

2.2 The denotational domain

By virtue of using a simply typed language, we can choose a suitable denotational domain for each type. This avoids having to solve recursive domain equations involving the function type (as in the untyped case), which we struggle with (see Sec. TODO).

The semantic domain for a type τ is a DCPO $\mathcal{D}(\tau)$, which is either bottom or a value $\mathcal{V}(\tau)$:

$$\begin{aligned}
\mathcal{D}(\tau) &= \mathcal{V}(\tau)_\bot\\
\mathcal{V}(\tau_1 \to \tau_2) &= [\mathcal{V}(\tau_1)_\bot \to_m \mathcal{D}(\tau_2)]\\
\mathcal{V}(\mathbf{Bool}) &= \mathbb{B}\\
\mathcal{V}(\mathbf{RBool}) &= \mathcal{F}\\
&\quad \text{where } \mathcal{F} = \mathbb{B}_\bot + (\mathcal{F}_\bot \times \mathcal{F}_\bot) + (\mathcal{F}_\bot \times \mathcal{F}_\bot)
\end{aligned}$$

Some notes:

(1) To stay close to lazy functional programming, every type is lifted. In particular, $\bot \sqsubset (\lambda x.\bot)$.
(2) The function type contains all *monotonic* functions (hence the m at the arrow), and not just the *continuous*, as usual. We'll soon see why we need this. I write f or $(\lambda v.f(v))$ for the elements in this domain.
It remains to be seen if this causes problems, and whether we can find a tighter model for the function space.
(3) The booleans are just $\mathbb{B} = \{\mathbf{f}, \mathbf{t}\}$, discretely ordered. I write b for an arbitrary element of $\mathbb{B}$.
(4) The denotation of **RBool** is $\mathcal{F}$, the (possibly infinite) formulas built from $\mathrm{Mk}(d)$, $\mathrm{And}(d,d)$ and $\mathrm{Or}(d,d)$.
(5) For some examples, it's useful to talk about an unary identity operation

$$\mathbf{id} :: \mathbf{RBool} \to \mathbf{RBool}$$

with denotation $\mathrm{Id}(d)$. Think of $\mathbf{id}\ e = \mathbf{and}\ x\ x$ resp. $\mathrm{Id}(d) = \mathrm{And}(d,d)$.

2.3 The denotation of expressions

Finally we can give the denotation of (well-typed) expressions: For every typing derivation $\Gamma \vdash e\colon \tau$ and well-typed environment $\rho \in \Pi_{x\in\mathrm{Var}}\mathcal{D}(\Gamma(x))$ we can deifne the dentotation $[\![e]\!]_\rho \in \mathcal{D}(\tau)$. The equations for the lambda calculus fragment, the booleans, and the **RBool** constructors are standard:

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x)\\
[\![\lambda x.e]\!]_\rho &= \lambda v.[\![e]\!]_{\rho\sqcup\{x\mapsto v\}}
\end{aligned}$$

$$\begin{aligned}
[\![e\ x]\!]_\rho &= \begin{cases} f(\rho(x)) & \text{if } [\![e]\!]_\rho = \lambda v. f(v) \\ \bot & \text{else} \end{cases} \\
[\![\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\!]_\rho &= [\![e_2]\!]_{\rho \sqcup \{x \mapsto \operatorname{lfp} h\}} \quad \text{where } h(d) = [\![e_1]\!]_{\rho \sqcup \{x \mapsto d\}} \\
[\![\mathbf{True}]\!]_\rho &= \mathbf{t} \\
[\![\mathbf{False}]\!]_\rho &= \mathbf{f} \\
[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]_\rho &= \begin{cases} [\![e_2]\!]_\rho & \text{if } [\![e_1]\!]_\rho = \mathbf{t} \\ [\![e_3]\!]_\rho & \text{if } [\![e_1]\!]_\rho = \mathbf{f} \\ \bot & \text{else} \end{cases} \\
[\![\mathbf{mk}]\!]_\rho &= \lambda v.\, \mathrm{Mk}(v) \\
[\![\mathbf{and}]\!]_\rho &= \lambda v_1 v_2.\, \mathrm{And}(v_1, v_2) \\
[\![\mathbf{or}]\!]_\rho &= \lambda v_1 v_2.\, \mathrm{Or}(v_1, v_2)
\end{aligned}$$

Here we can see that **and** and **or** are *lazy* and thus can be used to tie the knot, as described in the paper.

Because we allow monotonic (not continuous) functions in $\mathcal{D}(\tau_1 \to \tau_2)$, the semantics itself is monotonic (not continuous) in the environmet ρ. This means the least fixed-point $\operatorname{lfp} h$ in the semantics for **let** still exists, but may not necessarily be $\bigsqcup_i h^i(\bot)$.

All the magic is now in the semantics for **get**, which collects the formula constructed from Mk, And and Or, solves it in the two-point lattice $\mathscr{B} = \{\mathscr{t}, \mathscr{f}\}$ with $\mathscr{f} \sqsubset \mathscr{t}$ (and operations $\wedge$ and $\vee$), and returns the result as a $\mathbb{B}$.

More precisely, it uses a helper function

$$\mathit{interp} :: \mathcal{F}_\bot \to \mathscr{B}^\top$$

which is the smallest function that solves the equations

$$\begin{aligned}
\mathit{interp}(\mathrm{Mk}(b)) &= i(b) & &\text{where } i(\mathbf{t}) = \mathscr{t} \\
& & &\phantom{\text{where }} i(\mathbf{f}) = \mathscr{f} \\
& & &\phantom{\text{where }} i(\bot) = \top \\
\mathit{interp}(\mathrm{And}(d_1, d_2)) &= \mathit{interp}(d_1) \wedge \mathit{interp}(d_2) & &\text{if } \mathit{interp}(d_1), \mathit{interp}(d_2) \neq \top \\
\mathit{interp}(\mathrm{Or}(d_1, d_2)) &= \mathit{interp}(d_1) \vee \mathit{interp}(d_2) & &\text{if } \mathit{interp}(d_1), \mathit{interp}(d_2) \neq \top \\
\mathit{interp}(d) &= \top & &\text{for any other } d
\end{aligned}$$

Some observations about *interp*:

- The $\top$ value serves as an error indicator, when a the formula cannot be interpreted – here, if any value is unknown ($\bot$); but later we will see that it can be used to model when solving the equations does not terminate.
- This function is *not* monotonic from $\mathcal{F}_\bot$ to $\mathscr{B}^\top$, because $i \colon \mathbb{B}_\bot \to \mathscr{B}^\top$ isn't.
- Its definitional equations are monotonic in the recursive calls, so if you think of it as an infinite system of equations, indexed by the elements of $\mathcal{F}$, the definition becomes well-defined even for infinite formulas. For example

  $$\mathit{interp}(\mu d.\, \mathrm{And}(d, d)) = \mathscr{f}.$$

- We can only have $\mathit{interp}(d) \neq \top$ if d consist only of constructors, and in particular no $\bot$ occurs anywhere therein. Such a d is a *maximal* element of $\mathcal{F}$; there exists no strictly larger element.

With that we can define

$$[\![\mathbf{get}]\!]_\rho = \lambda v.e(\mathit{interp}(v)) \quad \text{where } e(\not{f}) = \mathbf{f}$$
$$e(\not{t}) = \mathbf{t}$$
$$e(\top) = \bot$$

More observations:

- Again, the mapping $e :: \mathscr{B}^\top \to \mathbb{B}_\bot$ therein is not monotonic.
- But – crucially – the the composition of e and *interp*, and thus the denotation of **get**, is monotonic:
  Assume $v_1 \sqsubset v_2$. So v_1 is not a maximal element, therefore $\mathit{interp}(v_1) = \top$, hence $[\![\mathbf{get}]\!]\ v_1 = \bot$, which suffices for $[\![\mathbf{get}]\!]\ v_1 \sqsubseteq [\![\mathbf{get}]\!]\ v_2$.
- The function is not continuous: Consider the sequence $d_i = \mathrm{Id}^i(\bot) \in \mathcal{F}$. It forms a chain
  $$\bot \sqsubset \mathrm{Id}(\bot) \sqsubset \mathrm{Id}(\mathrm{Id}(\bot)) \sqsubset \cdots$$
  with limit $d_\omega = (\mu d.\, \mathrm{Id}(d))$. We have
  $$[\![\mathbf{get}\ x]\!]\ d_i = \bot$$
  for all i, but
  $$[\![\mathbf{get}\ x]\!]\ d_\omega = \mathbf{f}.$$
  because $\mathit{interp}(d_\omega) = \not{f}$.
  We still get a well-defined semantics $[\![\cdot]\!]$; see the comment above about the existence of the least fixed point in the **let** semantics.
- This semantics is more defined that what we can implement, because *interp* works even in cases where we did not tie the knot. So we not only get
  $$[\![\mathbf{let}\ x = \mathbf{id}\ x\ \mathbf{in}\ \mathbf{get}\ x]\!] = \mathbf{f}$$
  as we expect, but also
  $$[\![\mathbf{let}\ x = (\lambda y.\mathbf{id}\ (x\ y))\ \mathbf{in}\ \mathbf{get}\ (x\ y)]\!] = \mathbf{f}$$
  which is not what we see in the implementation.
  The problem is that our semantics is call-by-name, not call-by-need, so we cannot distinguish the productive tied knot from the other expression.
  This can be fixed by elaborating the semantics along the lines of "Call-by-Need Is Clairvoyant Call-by-Value" and forcing $[\![\mathbf{get}]\!]\ d = \bot$ if d isn't knot-tied. (*I have TODO that in another section below.*)
  It seems that that semantics will simply be less defined than this one, but agree when they are both not bottom, so the call-by-name semantics may already useful for fast-and-lose reasoning.

2.4 Equational reasoning

Now that we have defined the semantics, can we use it?

2.4.1 Some things work. It seems we can do some amount of equational reasoning. Equations related to the lambda calculus like

$$[\![(\lambda x.e[x])\ y]\!] = [\![e[y]]\!]$$

hold as usual.

Moreover, we can derive program equations like

$$[\![\mathbf{get}\ (\mathbf{and}\ x\ y)]\!] = [\![(\mathbf{get}\ x)\&\&!(\mathbf{get}\ y)]\!]$$

where &&! a strict conjunction operator on $\mathbb{B}_\bot$:

If $\mathit{interp}(\rho(x)) = \top$ or $\mathit{interp}(\rho(y)) = \top$, then both sides are $\bot$. Else we can can calculate (using $\wedge$ both on $\mathscr{B}$ and $\mathbb{B}$)

$$\begin{aligned}
[\![\mathbf{get}\ (\mathbf{and}\ x\ y)]\!]_\rho &= [\![\mathbf{get}]\!]\ [\![\mathbf{and}\ x\ y]\!]_\rho \\
&= e(\mathit{interp}(\mathbf{and}(\rho(x), \rho(y)))) \\
&= e(\mathit{interp}(\rho(x)) \wedge \mathit{interp}(\rho(y))) \\
&= e(\mathit{interp}(\rho(x))) \wedge e(\mathit{interp}(\rho(y))) \\
&= [\![\mathbf{get}]\!]\ \rho(x) \wedge [\![\mathbf{get}]\!]\ \rho(y) \\
&= [\![\mathbf{get}\ x]\!]_\rho \wedge [\![\mathbf{get}\ y]\!]_\rho \\
&= [\![(\mathbf{get}\ x)\&\&!(\mathbf{get}\ y)]\!]_\rho
\end{aligned}$$

Since all moving parts are defined as fixed-points of one sort or another, equational reasoning works very well.

2.4.2 Some things do not work. Unfortunately, dome desirable identities like the following do not seem to hold

$$\begin{aligned}
[\![\mathbf{and}\ x\ y]\!] &= [\![\mathbf{and}\ y\ x]\!] \\
[\![\mathbf{or}\ x\ x]\!] &= [\![\mathbf{id}\ x]\!]
\end{aligned}$$

because the denotation D captures the full boolean formula, and not some denotation thereof, only to be interpreted by *interp*.

Can this be solved? Can we somehow inline the effect of *interp* into the denotational of **and**, without breaking the well-definedness of the semantics?

2.5 Other domains

The above isn't very specific to the boolean domain $\mathscr{B}$, and should work without much changes for other domains A such as $\mathcal{P}(\mathbb{N})$ or $\mathcal{P}_{\text{fin}}(\mathbb{N})$: All operations are modelled as constructors in D, and then *interp* interprets these formulas in $A^\top$.

This even does the right thing for non-complete domains such as $\mathcal{P}_{\text{fin}}(\mathbb{N})$: By adjoining the $\top$, and letting all operations map $\top$ to $\top$, a program like

```
let x = RS.insert 0 (RS.map (+1) x) in x
```

will be solved as $\top$ by *interp* and thus end up being $\bot$, as it should.

If we want to model `RMap a b`, where values from D are stored, but not looked at by *interp*, the argument for why $[\![\mathbf{get}]\!]$ is continuous is more involved, as we cannot simply argue via maximality, but may need some kind of parametricity argument.

2.6 Why not...

Let's briefly look at other attempts and variants, and where they failed.

2.6.1 A simpler domain. One might expect to be able to simply use the two-point lattice $\mathscr{B}$ directly as the domain for **RBool**, instead of keeping the formulas $\mathcal{F}$ around until we **get** them:

$$\begin{aligned}
\mathcal{D}(\mathbf{RBool}) &= \mathcal{V}(\mathbf{RBool})_\bot \\
\mathcal{V}(\mathbf{RBool}) &= \mathscr{B}
\end{aligned}$$

$$\llbracket \mathbf{mk} \rrbracket = \lambda v. \begin{cases} \mathit{f} & \text{if } v = \mathbf{f} \\ \mathit{t} & \text{if } v = \mathbf{t} \\ \bot & \text{if } v = \bot \end{cases}$$

$$\llbracket \mathbf{get} \rrbracket = \lambda v. \begin{cases} \mathbf{f} & \text{if } v = \mathit{f} \\ \mathbf{t} & \text{if } v = \mathit{t} \\ \bot & \text{if } v = \bot \end{cases}$$

$$\llbracket \mathbf{and} \rrbracket = \lambda v_1 v_2. \begin{cases} v_1 \wedge v_2 & \text{if } v_1, v_2 \neq \bot \\ \bot & \text{else} \end{cases}$$

However, now the denotation of expressions is not monotonic,

$$\mathit{f} \sqsubseteq \mathit{t} \quad \text{but} \quad \llbracket \mathbf{get}\ x \rrbracket_{\{x \mapsto \mathit{f}\}} = \mathbf{f} \not\sqsubseteq \mathbf{t} = \llbracket \mathbf{get}\ x \rrbracket_{\{x \mapsto \mathit{t}\}},$$

and the whole semantics is no longer well-defined – hence the need for something more elaborate.

2.6.2 An untyped domain. Often, denotational semantics are presented in an untyped way, with a single Domain D for all expressions, and operations failing (returning $\bot$) when used in an ill-typed way.

To do so in our case, we would start describe the domain D via a recursive domain equations, maybe like this:

$$D = ([D \rightarrow_m D] + \mathbb{B} + \mathcal{F})_\bot$$

where $\mathcal{F} = D + (D \times D) + (D \times D)$.

We'd now have to solve that domain equation, and prove that a CPO satisfying that equation actually exists. For the usual constructions, including the space of *continuous* functions, this is a well-known theorem. Our semantics, for better or worse, has to allow certain non-continuous functions to model the semantics for **get**. And once the domain equation recurses through $\rightarrow_m$, it is no longer generally the case that the domain equation has a solution.

We side-step the issue by focusing on the simply typed fragment, where we can assemble the semantic domain in a type-syntax driven way, without the need to recurse through $\rightarrow_m$.

3 TYING THE KNOT WITH LAZY EVALUATION

As mentioned above, our denotational semantics is too permissive: It assignes non-bottom meaning to programs we do not expect to run. This is because we only expect to handle those (infinite) formulas that arise from finite graphs in the heap, i.e. a tied knot, when evaluated lazily, but not others.

A denotational semantics that captures needs to be able to observe sharing; it must be a call-by-need semantics. How abstract can we be, while still capturing that difference? Can we still avoid talking about heaps and graphs? It seems we can, building on "Call-by-Need Is Clairvoyant Call-by-Value" by Hacket and Hutton.

Their goal was to find a semantics (operational and denotational) that captures the *cost* of evaluation in lazy evaluation, for example to prove improvement of program transformations. This is a bit more detail than we need, but, as a side-effect so to say, it allows us to recognize tied knots.

3.1 Cost counting domains

To apply their approach, we have to replace $D_\bot$ in our semantics domains with the more expressive

$$D_c := (\omega^{\mathrm{op}} \times D)_\bot$$

where ω^{op} is the set of natural numbers in reverse order ($\dots \sqsubset 2 \sqsubset 1 \sqsubset 0$). The intuition is that such a value can either denote something non-terminating ($\bot$) or a value ((n, v)) that remembers that it takes n steps to obtain that value. The reverse ordering on the costs means

$$\bot \sqsubset \dots \sqsubset (2, v) \sqsubset (1, v) \sqsubset (0, v)$$

and the usual "more defined" becomes "more defined or cheaper to calculate", which matches the intuition that $\bot$ takes an infinite number of steps.

According to an errata[1], the following order should be used on $\omega^{\text{op}} \times D$:

$$(k, f) \sqsubseteq (k', f') \iff k \geq k' \wedge \forall v, k \blacktriangleright f(v) \sqsubseteq k' \blacktriangleright f'(v)$$

which supposedly leads to

$$(k_1, f_1) \sqcup (k_2, f_2) = (k', \lambda v.(k_1 - k') \blacktriangleright f_1(v) \sqcup (k_2 - k') \blacktriangleright f_2(v)) \text{ where } k' = \min(k, k')$$

We need to extend this to data types, it seems. I am not yet sure what happens for binary ones, but for unary constructors like **id** I think this means we get

$$(k_1, \mathbf{id}(d_1)) \sqcup (k_2, \mathbf{id}(d_2)) = (k', \mathbf{id}((k_1 - k') \blacktriangleright d_1 \sqcup (k_2 - k') \blacktriangleright d_2)) \text{ where } k' = \min(k, k')$$

The operation $n \blacktriangleright \cdot$ adds a cost of n steps; we liberally use this operation at type $D_\bot \to D_c$ to add a cost annotation,

$$n \blacktriangleright d = (n, d),$$

and at type $D_c \to D_c$ to increase the cost,

$$n \blacktriangleright (m, d) = (n + m, d).$$

A value $d \in D$ can be considered a value of D_c as $(0, d)$. Thus in all cases, we have

$$\begin{aligned} n \blacktriangleright \bot &= \bot \\ 0 \blacktriangleright d &= d. \end{aligned}$$

Furthermore, for $n \neq 0$, $n \blacktriangleright d = d$ holds *only* for $d = \bot$. The operation $\pi_2(\bot) = \bot$, $\pi_2((c, v)) = v$ extracts the underlying value, throwing away the cost annotation.

3.2 Semantics of types

Our new semantic domains now become

$$\begin{aligned} \mathcal{D}(\tau) &= (\omega^{\text{op}} \times \mathcal{V}(\tau))_c \\ \mathcal{V}(\tau_1 \to \tau_2) &= [\mathcal{V}(\tau_1)_\bot \to_m \mathcal{D}(\tau_2)] \\ \mathcal{V}(\mathbf{Bool}) &= \mathbb{B} \\ \mathcal{V}(\mathbf{RBool}) &= \mathcal{F} \\ &\quad \text{where } \mathcal{F} = \mathbb{B}_c + (\mathcal{F}_c \times \mathcal{F}_c) + (\mathcal{F}_c \times \mathcal{F}_c) \end{aligned}$$

Note that the domain for function types is $[\mathcal{V}(\tau_1)_\bot \to_m \mathcal{D}(\tau_2)]$, so function *arguments* do not carry a cost; they are either bottom or already a value.

[1]https://www.cs.nott.ac.uk//~pszjlh/cbncbv_erratum.pdf

3.3 Semantics of expressions

We now have to add cost counting to our semantics function. The environment does not include costs $\rho \in \Pi_{x\in\mathrm{Var}}\mathcal{V}(\Gamma(x))_\bot$; the cost of a lazy binding, if it is going to be used, is accounted for at binding time.

But how is that possible, when under the call-by-need strategy a lazy binding becomes an unevaluated thunk first, and the first evaluation statefully updates it? This is solved by the titular clairvoyance: The denotation of `let x = e1 in e2` considers the semantics of both cases; one where `e1` is evaluated (and immediatelly accounted for), and one where it is simply dropped. Taking the better of the two denotations ($\sqcup$) elegantly does the right thing.

It will simplify our live considerably to only allow expressions in A-normal form (ANF), and expect the argument in an function application to always be a variable. This way, *only* the `let` construct deals with bindings:

$$
\begin{aligned}
[\![x]\!]_\rho &= 1 \blacktriangleright \rho(x)\\
[\![\lambda x.e]\!]_\rho &= \lambda v.[\![e]\!]_{\rho\sqcup\{x\mapsto v\}}\\
[\![e\;x]\!]_\rho &= \begin{cases}(1+n)\blacktriangleright f(\rho(x)) & \text{if } [\![e]\!]_\rho = n \blacktriangleright (\lambda v.f(v))\\ \bot & \text{else}\end{cases}\\
[\![\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\!]_\rho &= (1 \blacktriangleright [\![e_2]\!]_{\rho\sqcup\{x\mapsto\bot\}}) \sqcup\\
&\quad\begin{cases}(1+n)\blacktriangleright [\![e_2]\!]_{\rho\sqcup\{x\mapsto v\}} & \text{if } \mathrm{lfp}\,h = (n,v)\\ \bot & \text{else}\end{cases}\\
&\quad\text{where } h(d) = [\![e_1]\!]_{\rho_\sqcup\{x\mapsto\pi_2(d)\}}\\
[\![\mathbf{True}]\!]_\rho &= \mathbf{t}\\
[\![\mathbf{False}]\!]_\rho &= \mathbf{f}\\
[\![\mathbf{if}\,e_1\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3]\!]_\rho &= \begin{cases}(1+n)\blacktriangleright [\![e_2]\!]_\rho & \text{if } [\![e_1]\!]_\rho = n\blacktriangleright \mathbf{t}\\ (1+n)\blacktriangleright [\![e_3]\!]_\rho & \text{if } [\![e_1]\!]_\rho = n\blacktriangleright \mathbf{f}\\ \bot & \text{else}\end{cases}\\
[\![\mathbf{mk}]\!]_\rho &= \lambda v.\,\mathrm{Mk}(v)\\
[\![\mathbf{and}]\!]_\rho &= \lambda v_1 v_2.\,\mathrm{And}(v_1,v_2)\\
[\![\mathbf{or}]\!]_\rho &= \lambda v_1 v_2.\,\mathrm{Or}(v_1,v_2)
\end{aligned}
$$

We see that value forms do not incurs a cost, while all other constructs bump the cost by one. Furthermore, the costs of (strictly evaluated) subexpressions, like function left-hand-sides or scrutinees, are propagated.

The most interesting equation is the one for **let**: the denotation is the better ($\sqcup$) of two possible worlds: One where x is simply unused, so no cost is incurred and its value is $\bot$, and one where x will (eventually) be evaluated. In that case, the evaluation of e_1 better terminate. We are finding a least fixed point, as the meaning of e_2 may depend on the value of x. We even allow e_2 to access its own value without additional cost ($\pi_2(d)$) – this is the essence of sharing. But it must be the case that e_2 is lazy in x for this fixedpoint to reach a non-bottom result: If $h(\bot) = [\![e_1]\!]_{\rho_\sqcup\{x\mapsto\bot\}} = \bot$, then $\mathrm{lfp}\,h = \bot$.

In the examples below we do not always want to write functions in ANF. If we understand $e_1\ e_2$ with non-trivial argument as a short hand for $\mathbf{let}\, x = e_2\, \mathbf{in}\, e_1\ x$ with fresh x, we find

$$\begin{aligned}
[\![e_1\ e_2]\!]_\rho &= [\![\mathbf{let}\, x = e_2\, \mathbf{in}\, e_1]\!]_\rho \\
&= (1 \blacktriangleright [\![e_1\ x]\!]_{\rho\sqcup\{x\mapsto\bot\}}) \sqcup \begin{cases}(1+n_2) \blacktriangleright [\![e_1\ x]\!]_{\rho\sqcup\{x\mapsto v\}} & \text{if } [\![e_1]\!]_\rho = (n_2, v) \\ \bot & \text{else}\end{cases} \\
&= (2+n_1 \blacktriangleright f(\bot)) \sqcup \begin{cases}(2+n_1+n_2) \blacktriangleright f(v) & \text{if } [\![e_1]\!]_\rho = (n_2, v) \\ \bot & \text{else}\end{cases} \\
&= 2+n_1 \blacktriangleright \begin{cases} f(\bot) \sqcup n_2 \blacktriangleright f(v) & \text{if } [\![e_1]\!]_\rho = (n_2, v) \\ f(\bot) & \text{else}\end{cases} \\
&\quad \text{where } [\![e_1]\!]_\rho = n_1 \blacktriangleright (\lambda v. f(v))
\end{aligned}$$

3.4 Recognizing cyclic data structures

With this semantics, we can now distinguish knot-tied, cyclic data structure from other kinds of recursion. For example, we can distinguish $[\![\mathbf{let}\, x = \mathbf{id}\ x\, \mathbf{in}\, x]\!]$ from $[\![\mathbf{let}\, x = (\lambda y.\mathbf{id}\ (x\ y))\, \mathbf{in}\, x\ y]\!]$:

In the first case, we have

$$h(d) = [\![\mathbf{id}\ x]\!]_{\rho\sqcup x\mapsto\pi_2(d)} = 2 \blacktriangleright \mathrm{Id}(\pi_2(d))$$

so iterating h yields

$$\bot \sqsubset 1 \blacktriangleright \mathrm{Id}(\bot) \sqsubset 1 \blacktriangleright \mathrm{Id}(0 \blacktriangleright \mathrm{Id}(\bot)) \sqsubset \cdot$$

with limit $1 \blacktriangleright \mathrm{Id}(\mu d, 0 \blacktriangleright \mathrm{Id}(d))$, so alltogether

$$[\![\mathbf{let}\, x = \mathbf{id}\ x\, \mathbf{in}\, x]\!] = 3 \blacktriangleright \mathrm{Id}(\mu d.0 \blacktriangleright \mathrm{Id}(d)).$$

We see that the knot-tied data structure, after some finite outer cost, becomes an infinite tree with all costs 0. In other words: It takes a finite amount of steps to fully evaluate the (infinite) formula.

For the second expression, where no knot is tied, we have

$$\begin{aligned}
h(d) &= [\![\lambda y.\mathbf{id}\ (x\ y)]\!]_{\rho\sqcup\{x\mapsto\pi_2(d)\}} \\
&= \lambda v.1 \blacktriangleright [\![\mathbf{id}\ (x\ y)]\!]_{\rho\sqcup\{x\mapsto\pi_2(d), y\mapsto v\}} \\
&= \lambda v.1 \blacktriangleright [\![\mathbf{id}\ (x\ y)]\!]_{\rho\sqcup\{x\mapsto\pi_2(d), y\mapsto v\}} \\
&= \lambda v.3 \blacktriangleright \begin{cases}\mathrm{Id}(\bot) \sqcup n_2 \blacktriangleright \mathrm{Id}(v') & \text{if } [\![x\ y]\!]_{\rho\sqcup\{x\mapsto\pi_2(d), y\mapsto v\}} = (n_2, v') \\ \mathrm{Id}(\bot) & \text{else}\end{cases} \\
&= \lambda v.3 \blacktriangleright \mathrm{Id}\left(\begin{cases}(1+n_2) \blacktriangleright v' & \text{if } \pi_2(d) = f \text{ and } f(v) = (n_2, v') \\ \bot & \text{else}\end{cases}\right)
\end{aligned}$$

so

$$\begin{aligned}
h(\bot) &= \lambda v.3 \blacktriangleright \mathrm{Id}(\bot) \\
h(h(\bot)) &= \lambda v.3 \blacktriangleright (\mathrm{Id}(4 \blacktriangleright \mathrm{Id}(\bot))) \\
h(h(h(\bot))) &= \lambda v.3 \blacktriangleright (\mathrm{Id}(4 \blacktriangleright \mathrm{Id}(4 \blacktriangleright \mathrm{Id}(\bot)))) \\
\mathrm{lfp}\, h &= \lambda v.3 \blacktriangleright (\mathrm{Id}(\mu d.4 \blacktriangleright \mathrm{Id}(d)))
\end{aligned}$$

and we see that evaluating the whole forumula is *not* possible with finite cost.

3.5 An implementable get

This leads us to the right intution for the denotation of **get**: It can only return a non-bottom result if fully evaluating the formula is possible with a finite cost. Therefore, we make it add up all the costs involved. The function *cost* is the *smallest* function satisfying

$$\begin{aligned}
cost :: \mathcal{F}_c &\to \omega^\top \\
cost(n \blacktriangleright \mathrm{Mk}(d)) &= n \\
cost(n \blacktriangleright \mathrm{And}(d_1, d_2)) &= n + cost(d_1) + cost(d_2) \\
cost(n \blacktriangleright \mathrm{Or}(d_1, d_2)) &= n + cost(d_1) + cost(d_2) \\
cost(\bot) &= \top
\end{aligned}$$

We have $cost\, d \sqsubset \top$ only if d is a (possibly infinite) formula with no $\bot$ anywhere and finite cost annotations. In particular

$$cost(\mu v, 0 \blacktriangleright \mathrm{Id}(v)) = 0 \quad \text{and} \quad cost(\mu v, 1 \blacktriangleright \mathrm{Id}(v)) = \bot.$$

And finally the **get** operation now report this cost and, crucially, return $\bot$ if the cost is not finite:

$$[\![\mathbf{get}]\!]_\rho = \lambda v. \begin{cases} n \blacktriangleright e(interp(v)) & \text{if } cost(v) = n \\ \bot & \text{if } cost(v) = \top \end{cases}$$

Here, *interp* is like before, simply ignoring the cost annotations.

Is this still monotonic? Like before, the result non-bottom only if the argument is finite, free of bottoms and with finite total costs. A larger argument can thus only differ in the cost annotations, and have *smaller* numbers, which means the result of **get** will have a smaller cost number, which means it is larger, and all is well.