Denotational Semantics for Probabilistic Recurrences

by

Celeste Barnaby

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors in Mathematics and Computer Science

Acknowledgements

First, I would like to thank my research advisor, Norman Danner, who has consistently challenged and motivated me during my time at Wesleyan. He gave me my first opportunity to participate in research, for which I am very grateful. Thanks to Jim Lipton, Dan Licata, Saray Shai, and Danny Krizanc, who, along with Norman Danner, have taught wonderful computer science courses which I have had the pleasure of taking.

An additional thanks to my readers, Dan Licata and Saray Shai, for reading and evaluating my thesis.

Finally, thanks to Danny, Emmet, and Joanna, for being great housemates and friends, and to my mom, for being a great mom.

Abstract

Former work by Danner et al. [2015] has formalized a method of extracting recurrences that bound the complexity of higher-order programs. However, these bounds do not offer useful information about the costs of all programs—in particular, programs with stochastic processes. Karp [1994] defines several forms of recurrences describing the cost of probabilistic algorithms; we develop languages which allow us to express recurrences of these forms. We also offer denotational semantics for these languages, and demonstrate using example recurrences given by Karp that the denotation of a recurrence is equal to its solution. These languages and denotational semantics may be tied in to the framework developed by Danner et al. to offer more useful bounds on a wider array of programs.

Contents

CHAPTER 1

Introduction

Analyzing the complexity and performance of an algorithm is an important problem in computer science. We want our programs to run as quickly and as efficiently as possible; thus, it is crucial that we understand how much time and space it takes to execute a program, with respect to its input size. Prior work has been done on the development of tools for automating the analysis of program complexity, which would allow us to make these judgements more quickly and accurately.

However, such work has so far only offered a method for obtaining an upper bound on the worst-case runtime of an algorithm—which in certain cases does not reflect the actual runtime of an algorithm. In particular, this method does not provide useful bounds on algorithms with stochastic processes, where the runtime may be quite slow in the worst case but is, on average, much more efficient. Developing methods for extracting tighter upper-bounds on the costs of probabilistic algorithms would allow us to analyze a wider array of programs.

1. Automated Complexity Analysis

The cost of an algorithm is a function from the size of the input of the algorithm to the number of steps necessary to evaluate that input. As an example, consider the following `add` function, which adds an integer x to every element in a list ys of integers:

```
let rec add x ys =
match ys with
| [] -> []
| (y::ys') -> (y+x):: add x ys'
```

In a traditional cost analysis of this function, we assume that integer addition requires some constant number of steps c. The cost of this function is dependent upon the length of the list—that is, the more integers in the list, the more adding we need to do, and the longer the function will take to run. We may write a recurrence T describing its cost, based on the length n of the input list:

$$T(0) = 0$$

$$T(n) = c + T(n-1)$$

To determine the cost of `add`, we need only solve this recurrence. We find that $T(n) = cn$, meaning this function is linear in the length of the list.

This approach has several shortcomings. To start, it is an ad hoc way to analyze cost. We cannot generalize this result for use in other functions; thus, all traditional cost analyses involve writing recurrences by hand. Further, there is no formal connection between the algorithm and the recurrence, meaning we must manually verify that the recurrence we write down accurately describes the function we are analyzing. The combination of these issues creates a lot of potential for human error.

Another problem is that we assume that the work done to each element of the list requires a constant number of steps. This assumption is fine in the case of `add`, where all we are doing is adding integers; however, if we consider the more general `map` function, we run into some complications.

```
let rec map f ys =
match ys with
| [] -> []
| (y::ys') -> f(y):: map f ys'
```

Suppose ys is a list of lists of integers, and f is a function that performs selection sort on a single list. Then $f(y)$ does not take a constant amount of work; rather, its cost is quadratic in the length of y. In order to write a correct recurrence for `map`, we need some information about the cost of $f(y)$, where the size of y is variable. This issue becomes even thornier if the function f is a composition of functions $g \circ h$. In this case, we need information about the cost of $g(h(y))$, meaning we in turn need information about the *size* of $h(y)$— the list output by applying a list y to h.

Danner et al. [2015] aim to resolve these issues by developing a formal method for extracting recurrences that bound the complexity of higher-order programs. This work offers an extraction function that maps programs written in a language with structural list recursion— referred to as a source language— to a complexity. A complexity is a pair consisting of a cost— the steps required to evaluate the program—and a potential— the size of the output of this program. In addition, this work provides a bounding theorem which guarantees that this extracted complexity is a bound on the actual cost of evaluating the source language program.

The recurrences produced by the extraction function are all syntactic: they are cost-annotated source language programs, rather than what we would traditionally think of as a recurrence for an algorithm. Danner et al. define a denotational semantics for this complexity language, as a formal means of constructing a mathematical object to describe the meaning of programs in this language. The denotations of these cost-annotated programs resemble more familiar recurrences.

2. Probabilistic Recurrence Relations

Some algorithms have probabilistic elements which affect the way we analyze them. A prime example of this is randomized quicksort—the standard quicksort algorithm wherein the pivot is chosen to be a random element in the list. In the worst case, this algorithm is $O(n^2)$, making it no more efficient than other sorting algorithms such as bubble sort or insertion sort. However, in the average case it is $O(n \log n)$, making it one of the most efficient sorting algorithms. Hence, analyses of probabilistic algorithms must consider analyses of runtime other than worst case. One approach is to bound the upper tails of the probability distribution of the algorithm. This allows us to ensure, for instance, that the probability of quicksort taking much longer than $O(n \log n)$ is very small.

While there are proofs that derive the bounds of the cost of randomized quicksort, they commonly use imprecise, ad hoc arguments: that is, they cannot be applied to other algorithms with similar randomized elements. Karp [1994] offers a method for analyzing stochastic divide-and-conquer processes by describing their costs as recurrence relations of the form

$$T(x) = a(x) + T(h_1(x)) + \cdots + T(h_n(x))$$

where x is a non-negative real number describing the size of the input, $h_1(x)$ through $h_n(x)$ are random variables describing the sizes of the subproblems (e.g. in the case of quicksort, $h_1(x)$ and $h_2(x)$ describe the sizes of the two sublists), a is a function describing the work necessary to generate these subproblems, and $T(x)$ is a random variable describing the total running time of the algorithm on an input of size x. Karp then offers several methods for obtaining tight bounds on the upper tails of the probability distribution of $T(x)$.

Note, however, that the recurrence relation $T(x) = a(x) + T(h(x))$ has some clear flaws that emerge when we attempt to assign types to its expressions. Since $T(x)$ is supposed to be random variable over inputs of size x, $T(x)$ is a function of type $\Omega_x \to \mathbb{R}$, where Ω_x is the sample space of inputs of size x. Then T is essentially a function of type $\mathbb{R} \to (\Omega_x \to \mathbb{R})$. However, we see that T recursively takes $h(x)$ as an argument, even though $h(x)$ is described as a random variable with type $\Omega_x \to \mathbb{R}$, rather than $\mathbb{R}$. This type-checking error is likely a result of the informality of the recurrence description, and we are meant to understand that T takes not the random variable $h(x)$ as an argument, but the result of plugging an input $l \in \Omega_x$ into $h(x)$. Using this interpretation, we then have $T(x)(l) = a(x) + T(h(x)(l))$. But this is also problematic, since $T(x)(l)$, the result of plugging an input l into $T(x)$, is a real number, while $T(h(x)(l))$ is a random variable.

Taking this a step further, we could interpret this to mean $T(x)(l) = a(x) + T(h(x)(l))(l')$, where $l' \in \Omega_{h(x)(l)}$ is the derived subproblem. But acquiring such an l' requires us to define an additional function $\hat{h}(x)$ that takes an input l and returns the derived subproblem l'. Even worse, this function is of type $\Omega_x \to \Omega_{h(x)(l)}$ — that is, its type is dependent upon the random variable $h(x)$. Thus, we find that Karp's seemingly straightforward recurrence relation quickly becomes increasingly complicated when we attempt to type-check it. Other researchers (Tassarotti and Harper [2017]) have corroborated these observations.

3. Contributions of This Thesis

Karp offers the equation $T'(x) = a(x) + T'(m(x))$ as a deterministic counterpart of a recurrence relation $T(x) = a(x) + T(h(x))$, where, for all x, $m(x)$ is equal to the upper bound of $h(x)$. This thesis defines languages that allow us

to write and type-check both probabilistic recurrences of the form T', and deterministic recurrences of the form T. These languages comprise a simple expression language including real number constants, identifiers, basic arithmetic operations, booleans, and application. These languages must also provide a way to express recursive functions, since this work centers around expressing recurrence relations.

In Chapter 2 we will define a language and denotational semantics that formalizes deterministic recurrences. A natural approach would be to use a general fixed-point construction and CPO semantics; however, this turns out to result in some unexpected problems. Instead, we define a language that has a more restrictive recursion construct `rec`, which we interpret using infinite sums.

While Karp proves theorems about recurrences of the form $T(x) = a(x) + T(m(x))$, the recurrences that come up in his examples of applications take a different form. Thus in Chapter 3, we formulate a language for those recurrences, where we are in fact able to use a general fixed-point construction and CPO semantics. We then investigate the difficulty of reconciling these two seemingly different forms of recursive definitions in Chapter 4.

Of course, the main point of Karp's paper is probabilistic recurrences, so we define a language and semantics for them in Chapter 5. This turns out to be quite complex: as discussed above, the typing of these recurrences itself is already problematic. Our solution is to support dependent types in our language. Since it proves to be quite difficult to combine dependent types with CPOs, we focus on a semantics that corresponds to the infinite sum interpretation for deterministic recurrences. We conclude in Chapter 6 with a discussion of how your work fits into Danner et al.'s recurrence extraction framework and further work.

CHAPTER 2

Deterministic Recurrences as Infinite Sums

[Karp, 1994, §1] states that the cost of recursive, stochastic algorithms may be described by recurrence relations of the form $T(x) = a(x) + T(h(x))$, where x is a non-negative real number, $h(x)$ and $T(x)$ are random variables, and a is a non-negative, real-valued function. He then offers a deterministic counterpart to this probabilistic recurrence, $T'(x) = a(x) + T'(m(x))$, where m is a real-valued function such that for all $x \in \mathbb{R},\ E(h(x)) \leq m(x), 0 \leq m(x)$, and $m(x), m(x)/x$ are nondecreasing. He asserts that when T' has a non-negative solution, it has a least solution u given by $u(x) = \sum_{i=1}^{\infty} a(m^i(x))$, where m^i is the i^{th} iterate of m.

In this chapter, we will define an expression language that allows us to write and type-check deterministic recurrences such as T'. Further, we will define a denotational semantics for this language, such that the interpretations of a recurrence is equal to its solution.

Consider that, in order to describe recurrence relations, our language must have some way of defining recursive functions. A first attempt at this involves defining a PCF-like language with a `real` type and a `fix` operator that assigns the least fixed-point to continuous functions using the CPO fixed-point theorem (the details of such a language are described further in chapter 3). Our denotational semantics then interprets all base types as flat-ordered CPOs—for instance, `real` interprets to $R_\perp$, the reals with a bottom element $\perp$, with each real number comparable only to $\perp$. Following standard PCF, arrow types $\tau \to \sigma$ interpret to a CPO whose bottom element is a function $\perp\colon \tau \to \sigma$ such that $\perp\,(x) = \perp_\sigma, \forall x \in \tau$.

A recursively-defined function f is interpreted as $\bigvee_n \{F^n \perp\}$, where $F : (\texttt{real} \to \texttt{real}) \to (\texttt{real} \to \texttt{real})$ is derived from the recursive definition.

However, this approach proves problematic, due to the fact that the only obvious interpretation of the base operations is strict—that is, it does not make sense for the sum of $\perp$ and another element to be anything other than $\perp$. The function F used to interpret a recurrence such as $T(x) = a(x) + T(m(x))$ has the form $F(g)(x) = a(x) + g(m(x))$. But if addition is strict, then

$$F(\perp)(x) = a(x) + \perp (px) = a(x) + \perp = \perp$$

so by induction we may show that for all n, $F^n(\perp)(x) = \perp$, so $\bigvee\{F^n(\perp)\}_{n=1}^{\infty} = \perp$. This result, is, clearly, not a solution to the recurrence $T(x) = a(x) + T(px)$, making this denotational semantics unsuitable.

A second approach involves altering the interpretation of the `real` type to $\mathbb{R}_{[0,+\infty]}$—the non-negative, extended reals. See that $\mathbb{R}_{[0,+\infty]}$ is a CPO: 0 is a bottom element, and any directed subset of $\mathbb{R}_{[0,+\infty]}$ must have a least upper bound. Such an interpretation resolves the issue of all recursive functions interpreting to $\perp$, because there is now a reasonable, non-strict interpretation of the base functions—e.g., $x + 0 = x$. In the above example, for instance, we would have

$$F(\perp)(x) = a(x) + \perp (px) = a(x) + 0 = a(x)$$

$$F(F(\perp))(x) = a(x) + F(\perp)(px) = a(x) + a(px)$$

so, by induction, $\forall n$, $F^n(\perp)(x) = \sum_{i=1}^{n} a(p^{i-1}x)$, and hence $\bigvee\{F^n(\perp)\}_{n=1}^{\infty}(x) = \sum_{i=1}^{\infty} a(p^{i-1}x)$. This *is* the solution to the recurrence $T(x) = a(x) + T(px)$, suggesting that this is the appropriate denotational semantics to use. However, this causes unexpected problems. Note that the standard interpretation of the equality

operator is as follows:

$$[\![e_1 = e_2]\!]\eta = \begin{cases} true \text{ if } ([\![e_1]\!]\eta = [\![e_2]\!]\eta \neq \bot) \\ false \text{ if } ([\![e_1]\!]\eta \neq [\![e_2]\!]\eta, [\![e_1]\!]\eta \neq \bot, [\![e_2]\!]\eta \neq \bot) \\ \bot \text{ otherwise} \end{cases}$$

with similar rules for $<, >, \leq, \geq$. But if our bottom element of $[\![\texttt{real}]\!]$ is 0, this prohibits us from writing a program where we compare a number to 0. This is too restrictive, as there are many divide-and-conquer algorithms where we continue a process on the condition that, say, an integer is non-negative, or a list is non-empty. Then we have to rewrite our interpretation of such expressions as

$$[\![e_1 = e_2]\!]\eta = \begin{cases} true \text{ if } [\![e_1]\!]\eta = [\![e_2]\!]\eta \\ false \text{ if } [\![e_1]\!]\eta \neq [\![e_2]\!]\eta \end{cases}$$

To see why this is a problem, consider the expression $x = 0$ and the chain $\{n\}_{n=0}^{\infty} \in [\![\texttt{real}]\!]$. See, then, that

$$[\![x = \texttt{0}]\!]\{x \mapsto 0\} = true$$

$$[\![x = \texttt{0}]\!]\{x \mapsto 1\} = false$$

$$[\![x = \texttt{0}]\!]\{x \mapsto 2\} = false$$

$$\ldots$$

so, since the standard interpretation of $[\![\texttt{bool}]\!]$ is flat-ordered—and there is no obvious alternative interpretation—$\{[\![x = 0]\!]\{x \mapsto n\}\}_{n=0}^{\infty}$ has no supremum. However, as we will see in chapter 3, in order to use the CPO fixed-point theorem

it will be necessary to show that for all expressions e and chains $a_0 \leq a_1 \leq \ldots$,

$$[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\}$$

Thus, this approach will not work either. In light of this, we define the a language that replaces the general fixed-point operator with a special-purpose recursive operator. Typing and equational semantics are given in Figures 1 and 2.

$$\begin{array}{rcl} \tau & ::= & \texttt{real} \mid \texttt{bool} \mid \tau \times \tau \mid \tau \to \tau \\ e & ::= & x \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \lambda x.e \mid e\ e \mid e + e \mid e - e \mid e * e \mid e/e \mid \texttt{true} \mid \texttt{false} \mid \\ & & e = e \mid e < e \mid e > e \mid e \leq e \mid e \geq e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid (e, e) \mid \\ & & \texttt{floor}(e) \mid \texttt{rec}(\lambda x.e, \lambda x.e) \end{array}$$

The `rec` operator allows us to write recurrences of the form $T(x) = a(x) + T(m(x))$, where T, a, and m are real-valued functions. The recurrence for T corresponds to the expression $\texttt{rec}(\lambda x.a, \lambda x.m)$. As seen in the typing rules, `rec` expressions describe recurrences only at type `real`. While this is a very restrictive subset of recursive functions, it allows us to express the recurrences described by Karp, while avoiding the problems of the previous approaches.

The denotational semantics for this language is given in Figures 3 and 4. Note, in these figures, that Γ is a partial function mapping variables to types, and a Γ-environment η is a partial function that maps a variable x to a value in the denotation of $\Gamma(x)$. Note, also, that the denotation of `real`, is $\mathbb{R} \cup \{\infty\}$. This is because we may write a recurrence of the form $T(x) = a(x) + T(m(x))$ whose only solution is ∞—see that this is true for $T(x) = 1 + T(x)$. Figure 4 defines $[\![\Gamma \vdash e : \tau]\!]\eta$ for a Γ-environment η, but we elide mentions of Γ and τ for

$$\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{real}}, \quad \frac{}{\Gamma \vdash \mathtt{1} : \mathtt{real}}, \ldots$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}, \quad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x : \sigma).e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{real} \quad \Gamma \vdash e_2 : \mathtt{real}}{\Gamma \vdash e_1 \circ e_2 : \mathtt{real}}, \circ \in \{+, -, *, /\}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{real} \quad \Gamma \vdash e_2 : \mathtt{real}}{\Gamma \vdash e_1 \circ e_2 : \mathtt{bool}}, \circ \in \{=, <, >, \geq, \leq\}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \mathtt{real}}{\Gamma \vdash \mathtt{floor}(e) : \mathtt{real}}$$

$$\frac{\Gamma \vdash \lambda x.a : \mathtt{real} \to \mathtt{real} \quad \Gamma \vdash \lambda x.m : \mathtt{real} \to \mathtt{real}}{\Gamma \vdash \mathtt{rec}(\lambda x.a, \lambda x.m) : \mathtt{real} \to \mathtt{real}}$$

FIGURE 1. Typing for the language.

convenience and just write $[\![e]\!]\eta$. Type-soundness is straightforward; thus, proof of the following theorem is omitted.

THEOREM 1. *If* $\Gamma \vdash e : \tau$, *then* $[\![\Gamma \vdash e : \tau]\!]\eta \in [\![\tau]\!]$.

We will prove equational soundness of our denotational semantics, in order to ensure that our interpretation of recurrences reflects their equational meaning.

$e = e$

$\mathtt{0} + \mathtt{0} = \mathtt{0},\ \mathtt{0} + \mathtt{1} = 1, \ldots,\ \mathtt{3} + \mathtt{5} = \mathtt{8}, \ldots$
equivalent rules for $-$, $*$, and / operations.

$(n = n) = \mathtt{true}$
$(n = m) = \mathtt{false}$, provided $n,\ m$ distinct numerals.

$\mathtt{if}\ \ \mathtt{true}\ \ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 = e_1$
$\mathtt{if}\ \ \mathtt{false}\ \ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 = e_2$

$\lambda x.e = \lambda y.[x \mapsto y]e$, provided y not free in e.

$\lambda x.e_1\ e_2 = [x \mapsto e_2]e_1$
$\mathtt{rec}(\lambda x.a, \lambda x.m) = \lambda x.(a + \mathtt{rec}(\lambda x.a, \lambda x.m)(m))$

FIGURE 2. Equational semantics for the language.

$$\begin{aligned}
[\![\mathtt{real}]\!] &= \mathbb{R}_\infty \\
[\![\mathtt{bool}]\!] &= \{true, false\} \\
[\![\tau \times \sigma]\!] &= [\![\tau]\!] \times [\![\sigma]\!] \\
[\![\tau \to \sigma]\!] &= \{f : [\![\tau]\!] \to [\![\sigma]\!] \mid\ f \text{ is continuous}\}
\end{aligned}$$

FIGURE 3. Denotational semantics for types.

THEOREM 2. *If* $\Gamma \vdash e_0 : \tau$ *and* $\Gamma \vdash e_1 : \tau$ *and* $e_0 = e_1$, *then for all* Γ*-environments* η,

$$[\![\Gamma \vdash e_0]\!]\eta = [\![\Gamma \vdash e_1]\!]\eta$$

$$
\begin{aligned}
[\![\texttt{0}]\!]\eta = 0,\ [\![\texttt{1}]\!]\eta = 1,\ \ldots \\
[\![x : \tau]\!]\eta &= \eta(x) \\
[\![\lambda(x:\tau).(e:\sigma)]\!]\eta &= f : [\![\tau]\!] \to [\![\sigma]\!] \\
\text{s.t. } \forall d \in [\![\tau]\!], f(d) &= [\![e]\!]\eta\{x \mapsto d\} \\
[\![e_1\ e_2]\!]\eta &= [\![e_1]\!]\eta([\![e_2]\!]\eta) \\
[\![e_1 + e_2]\!]\eta &= [\![e_1]\!]\eta + [\![e_2]\!]\eta \\
[\![e_1 - e_2]\!]\eta &= [\![e_1]\!]\eta - [\![e_2]\!]\eta \\
[\![e_1 * e_2]\!]\eta &= [\![e_1]\!]\eta * [\![e_2]\!]\eta \\
[\![e_1/e_2]\!]\eta &= [\![e_1]\!]\eta/[\![e_2]\!]\eta \\
[\![\texttt{true}]\!]\eta = true,\ [\![\texttt{false}]\!]\eta &= false \\
[\![e_1 = e_2]\!]\eta &= \begin{cases} true \text{ if } [\![e_1]\!]\eta = [\![e_2]\!]\eta \\ false \text{ if } [\![e_1]\!]\eta \neq [\![e_2]\!]\eta \end{cases}
\end{aligned}
$$

Similar rules for $<,\ \leq, >$ and $\geq$

$$
\begin{aligned}
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\eta &= \begin{cases} [\![e_2]\!]\eta \text{ if } [\![e_1]\!]\eta = true \\ [\![e_3]\!]\eta \text{ if } [\![e_1]\!]\eta = false \end{cases} \\
[\![(e_1, e_2)]\!]\eta &= ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![\texttt{floor}(e)]\!]\eta &= \lfloor [\![e]\!]\eta \rfloor \\
[\![\texttt{rec}(\lambda x.a, \lambda x.m)]\!]\eta &= rec : \mathbb{R}_\infty \to \mathbb{R}_\infty \text{ s.t. } \forall d \in \mathbb{R}_\infty, \\
rec(d) &= \sum_{i=0}^{\infty} [\![\lambda x.a]\!]\eta(([\![\lambda x.m]\!]\eta)^i(d))
\end{aligned}
$$

FIGURE 4. Denotational semantics for expressions.

PROOF. We will verify only the equation for `rec` expressions, as all others are trivial. Note that for all $d \in \mathbb{R}_\infty$,

$$
\begin{aligned}
[\![\lambda x.(a + \texttt{rec}(\lambda x.a, \lambda x.m)(m))]\!]\eta(d) = \\
&[\![a]\!]\{x \mapsto d\} + [\![\texttt{rec}(\lambda x.a, \lambda x.m)(m)]\!]\eta\{x \mapsto d\} \\
&= [\![\lambda x.a]\!](d) + [\![\texttt{rec}(\lambda x.a, \lambda x.m)]\!]\eta([\![m]\!]\eta\{x \mapsto d\}) \\
&= [\![\lambda x.a]\!](d) + [\![\texttt{rec}(\lambda x.a, \lambda x.m)]\!]\eta([\![\lambda x.m]\!](d)) \\
&= [\![\lambda x.a]\!](d) + \sum_{i=0}^{\infty} [\![\lambda x.a]\!]\eta(([\![\lambda x.m]\!]\eta)^i(d))
\end{aligned}
$$

$$= [\![\lambda x.a]\!](d) + \sum_{i=1}^{\infty} [\![\lambda x.a]\!]\eta(([\![\lambda x.m]\!]\eta)^i(d))$$

$$= \sum_{i=0}^{\infty} [\![\lambda x.a]\!]\eta(([\![\lambda x.m]\!]\eta)^i(d))$$

$$= [\![\texttt{rec}(\lambda x.a, \lambda x.m)]\!]\eta(d)$$

□

We want to gain some intuition that the denotation of a $\texttt{rec}(a, m)$ expression is equal to the solution of the corresponding recurrence, $T(x) = a(x) + T(m(x))$. Consider this example Karp offers, wherein $m(x) = px$, with p a positive constant less than 1, and $a(x) = 0,\ x < 1,\ a(x) = 1,\ x \geq 1$. Then $T(x) = a(x) + T(px)$, so by a simple inductive argument we can show that

$$T(x) = \begin{cases} 0 \text{ if } x < 1 \\ k \text{ if } \dfrac{1}{p^{k-1}} \leq x < \dfrac{1}{p^k} \end{cases}$$

We express this function in our language as follows:

$$\texttt{T} = \texttt{rec}(\lambda x.(\texttt{if } x < \texttt{1 then 0 else 1}), \lambda x.\texttt{p} * x)$$

We want to show that the interpretation of this expression has the same solution as the recurrence described by Karp. Let $\texttt{a} = \lambda x.\texttt{if } x < 1 \texttt{ then } 0 \texttt{ else } 1$, $\texttt{m} =$

$\lambda x.\mathtt{p} * x$, and see that

$$\begin{aligned}
[\![\mathtt{a}]\!] &= a : \mathbb{R} \to \mathbb{R} \text{ s.t. } a(d) = 0,\ d < 1, a(d) = 1,\ d \geq 1 \\
[\![\mathtt{m}]\!] &= m : \mathbb{R} \to \mathbb{R} \text{ s.t } \forall d \in \mathbb{R},\ m(d) = pd \\
[\![\mathtt{T}]\!] &= [\![\mathtt{rec}(\mathtt{a}, \mathtt{m})]\!] \\
&= rec : \mathbb{R} \to \mathbb{R} \text{ s.t. } \forall d \in \mathbb{R}, \\
rec(d) &= \sum_{i=0}^{\infty} [\![\mathtt{a}]\!]([\![\mathtt{m}]\!]^i(d)) \\
&= \sum_{i=0}^{\infty} a(m^i(d)) \\
&= \sum_{i=0}^{\infty} a(p^i(d))
\end{aligned}$$

Note that, for all $d < 1 \in \mathbb{R},\ [\![\mathtt{T}]\!](d) = 0$ and for all $d \geq 1 \in \mathbb{R}$, since $p < 1,\ \exists k \in \mathbb{N}$ s.t $\frac{1}{p^{k-1}} \leq d < \frac{1}{p^k}$. Then $p^i d < 1$ for $i \geq k$, so

$$\begin{aligned}
[\![\mathtt{T}]\!](d) &= \sum_{i=0}^{\infty} a(p^i(d)) \\
&= \sum_{i=0}^{k-1} a(p^i(d)) + \sum_{i=k}^{\infty} a(p^i(d)) \\
&= k + 0 = k
\end{aligned}$$

Therefore, $[\![\mathtt{T}]\!] = T$.

CHAPTER 3

Deterministic Recurrences as Fixed Points

[Karp, 1994, §2] offers several example applications of his cost-bounding theorems. However, the recurrences he offers here do not follow the definition he set forth in the prior section. For example, a recurrence for a randomized list ranking algorithm is written as

$$T(1) = 1$$

$$T(n) = 1 + T(3/4n).$$

Note that this recurrence only makes sense for inputs n such that $(3/4)^i n = 1$ for some i. Thus, we interpret the actual meaning of the general recurrence to be $T(n) = 1 + T(\lfloor 3/4n \rfloor)$.

The presence of an initial condition here is confusing: Karp has made no mention of initial conditions up to this point, and there is no clear way to write the above equations as one equation of the form $T(x) = a(x) + T(m(x))$. Simply put, this recurrence does not seem to be within the domain of recurrences Karp describes in section 1. Thus, we have no way of writing this recurrence using the language defined in the previous chapter.

In this chapter, we define a language which allows us to write and type-check the recurrence relations offered in section 2 of Karp's paper. The syntax of this language is mostly analogous to Programming Computable Functions (PCF) —a typed functional language with a fixed-point combinator— with one exception: instead of a `nat` type, we have a `real` type, as this is the domain of the functions

Karp discusses in this section. We will circumvent the issues we had with this approach in the previous chapter by writing these recurrences using conditional statements $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, with e_2 corresponding to the initial condition and e_3 to the general condition. We will see that this approach will allow us to have strict evaluation of base operations, while still obtaining correct solutions to recurrences.

$$
\begin{aligned}
\tau &::= \texttt{real} \mid \texttt{bool} \mid \tau \times \tau \mid \tau \to \tau \\
e &::= x \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \lambda x.e \mid e\ e \mid e + e \mid e - e \mid e * e \mid e/e \mid \texttt{true} \mid \texttt{false} \mid \\
&\quad\ e = e \mid e < e \mid e > e \mid e \leq e \mid e \geq e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid (e, e) \mid \\
&\quad\ \texttt{floor} \mid \texttt{fix}(\lambda f.\lambda x.e)
\end{aligned}
$$

The equational semantics of our language is given in Figure 2, and the denotational semantics is given in figures 3 and 4. This semantics is the standard one for PCF, where each type interprets to a complete partially ordered set (CPO). A CPO is a set S with two defining properties. Before we state these properties, we first offer some definitions related to ordered sets. An element $a \in S$ is called the bottom element (denoted by $\perp$) if for all $s \in S,\ a \leq s$. A subset $C \subset S$ is called a chain if $C = \{c_1, c_2, c_3, \ldots\}$, where $c_1 \leq c_2 \leq c_3 \leq \ldots$. An upper bound on C is an element $a \in S$ such that for all $i,\ c_i \leq a$. The least upper bound $\bigvee C$ is an upper bound of C such that, for any upper bound a of $C,\ \bigvee C \leq a$.

With these definitions in mind, we say S is a CPO if

(1) S has a bottom element $\perp$.

(2) For any chain $C \subset S,\ \bigvee C \in S$.

Additionally, if P and Q are CPOs, we say a function $f : P \to Q$ is continuous if, for every chain $C \subset P$, $f(\bigvee C) = \bigvee f(C)$.

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{real}}, \quad \frac{}{\Gamma \vdash \texttt{1} : \texttt{real}}, \ldots$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}, \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x : \sigma).e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \circ e_2 : \texttt{real}}, \circ \in \{+, -, *, /\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \circ e_2 : \texttt{bool}}, \circ \in \{=, <, >, \geq, \leq\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \texttt{real}}{\Gamma \vdash \texttt{floor}(e) : \texttt{real}}$$

$$\frac{\Gamma \vdash \lambda f.\lambda x.e : (\tau \to \sigma) \to (\tau \to \sigma)}{\Gamma \vdash \texttt{fix}(\lambda f.\lambda x.e) : \tau \to \sigma}$$

FIGURE 1. Typing for the language.

We will interpret `fix` expressions using the CPO fixed-point theorem. This theorem states that, for a CPO P and a continuous, order-preserving map $\Phi : P \to P$, the least fixed-point of Φ exists and is equal to $\bigvee \Phi^n(\bot)$.

In order to use this theorem on the interpretation of $\texttt{fix}(\lambda f.\lambda x.e) : \tau \to \sigma$ expressions, we must first prove the following.

$e = e$

$\mathtt{0} + \mathtt{0} = \mathtt{0},\ \mathtt{0} + \mathtt{1} = 1, \ldots,\ \mathtt{3} + \mathtt{5} = \mathtt{8}, \ldots$
equivalent rules for $-$, $*$, and / operations.

$(n = n) = \mathtt{true}$
$(n = m) = \mathtt{false}$, provided $n,\ m$ distinct numerals.

$\mathtt{if}\ \ \mathtt{true}\ \ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 = e_1$
$\mathtt{if}\ \ \mathtt{false}\ \ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 = e_2$

$\lambda x.e = \lambda y.[x \mapsto y]e$, provided y not free in e.

$\lambda x.e_1\ e_2 = [x \mapsto e_2]e_1$
$\mathtt{fix}(\lambda f.\lambda x.e) = [f \mapsto \mathtt{fix}(\lambda f.\lambda x.e)](\lambda x.e)$

FIGURE 2. Equational Semantics for the language.

$$[\![\mathtt{real}]\!] = \mathbb{R}_\perp$$
$$[\![\mathtt{bool}]\!] = \{true, false\}_\perp$$
For any set $S,\ S_\perp = S \cup \{\perp\},$ where $\forall x, y \in S,\ x \leq y \iff x = \perp$.
This is called the flat ordering of S.
$$[\![\tau \times \sigma]\!] = [\![\tau]\!] \times [\![\sigma]\!]$$
For all $(a, b),\ (c, d) \in [\![\tau \times \sigma]\!]$, let $(a, b) \leq (c, d)$ if and only if $a \leq c$ and $b \leq d$.
$$[\![\tau \to \sigma]\!] = \{f : [\![\tau]\!] \to [\![\sigma]\!]\ :\ f \text{ is continuous}\}$$
For all $f,\ g \in [\![\tau \to \sigma]\!]$, let $f \leq g$ if and only if $\forall x \in [\![\tau]\!],\ f(x) \leq g(x)$

FIGURE 3. Denotational semantics for types.

THEOREM 3. *For all types τ, $[\![\tau]\!]$ is a CPO — that is, $[\![\tau]\!]$ is an ordered set with a bottom element, such that for all $C \subset [\![\tau]\!]$, if C is a chain, then C has a least upper bound in $[\![\tau]\!]$.*

PROOF. by induction on the structure of τ.

Base Cases:

- $\tau = \mathtt{real}$

$$
\begin{aligned}
[\![\texttt{0}]\!]\eta = 0,\ [\![\texttt{1}]\!]\eta = 1,\ \ldots \\
[\![x : \tau]\!]\eta &= \eta(x) \\
[\![\lambda(x : \tau).(e : \sigma)]\!]\eta &= f : [\![\tau]\!] \to [\![\sigma]\!] \text{ s.t. } \forall d \in [\![\tau]\!], f(d) = [\![e]\!]\eta\{x \mapsto d\} \\
[\![e_1\ e_2]\!]\eta &= [\![e_1]\!]\eta([\![e_2]\!]\eta) \\
[\![e_1 + e_2]\!]\eta &= [\![e_1]\!]\eta + [\![e_2]\!]\eta \\
[\![e_1 - e_2]\!]\eta &= [\![e_1]\!]\eta - [\![e_2]\!]\eta \\
[\![e_1 * e_2]\!]\eta &= [\![e_1]\!]\eta * [\![e_2]\!]\eta \\
[\![e_1/e_2]\!]\eta &= \begin{cases} \bot \text{ if } [\![e_2]\!]\eta = 0 \\ [\![e_1]\!]\eta/[\![e_2]\!]\eta \text{ otherwise} \end{cases} \\
[\![\texttt{true}]\!]\eta = true,\ [\![\texttt{false}]\!]\eta &= false \\
[\![e_1 = e_2]\!]\eta &= \begin{cases} true \text{ if } ([\![e_1]\!]\eta = [\![e_2]\!]\eta \neq \bot) \\ false \text{ if } ([\![e_1]\!]\eta \neq [\![e_2]\!]\eta, [\![e_1]\!]\eta \neq \bot, [\![e_2]\!]\eta \neq \bot) \\ \bot \text{ otherwise} \end{cases}
\end{aligned}
$$

Similar rules for $<,\ \leq, >, \geq$

$$
\begin{aligned}
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\eta &= \begin{cases} [\![e_2]\!]\eta \text{ if } [\![e_1]\!]\eta = true \\ [\![e_3]\!]\eta \text{ if } [\![e_1]\!]\eta = false \\ \bot \text{ otherwise} \end{cases} \\
[\![(e_1, e_2)]\!]\eta &= ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![\texttt{floor}(e)]\!]\eta &= \lfloor [\![e]\!]\eta \rfloor \\
[\![\texttt{fix}(\lambda f.\lambda x.e)]\!]\eta &= fix[\![\lambda f.\lambda x.e]\!]\eta \\
&\text{where } fix \text{ assigns the least fixed point to continuous functions}
\end{aligned}
$$

FIGURE 4. Denotational semantics for expressions.

- $\tau = \texttt{bool}$

  In these cases, $[\![\tau]\!]$ is a set with flat ordering, and is thus a CPO.

Inductive Cases:

- $\tau = \tau_1 \times \tau_2$

  Note $[\![\tau]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]$. By inductive hypothesis, $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ are CPOs, and thus have bottom elements $\bot_1$ and $\bot_2$, respectively. Thus $(\bot_1, \bot_2)$ is a bottom element of τ.

Let $C \subset [\![\tau]\!]$ be a chain, and define

$$C_1 = \{x \in [\![\tau_1]\!] \mid \exists y \in [\![\tau_2]\!] \text{ s.t. } (x, y) \in C\},$$
$$C_2 = \{y \in [\![\tau_2]\!] \mid \exists x \in [\![\tau_1]\!] \text{ s.t. } (x, y) \in C\}.$$

Then $C_1 \subset [\![\tau_1]\!]$ and $C_2 \subset [\![\tau_2]\!]$ are chains, so $\bigvee C_1$ and $\bigvee C_2$ exist. We claim that $(\bigvee C_1, \bigvee C_2)$ is the least upper bound of C.

Let $(c_1, c_2) \in C$. Then $c_1 \in C_1$ and $c_2 \in C_2$, so $c_1 \leq \bigvee C_1$ and $c_2 \leq \bigvee C_2$. Thus, $(c_1, c_2) \leq (\bigvee C_1, \bigvee C_2)$, so $(\bigvee C_1, \bigvee C_2)$ is an upper bound of C.

Now let $(x, y) \in [\![\tau]\!]$ be an upper bound of C. Then $\forall c_1 \in C_1,\ c_1 \leq x$, so x is an upper bound of C_1. Similarly, y is an upper bound of C_2. Thus, it must be that $\bigvee C_1 \leq x$ and $\bigvee C_2 \leq y$, so $(\bigvee C_1, \bigvee C_2) \leq (x, y)$. Therefore, $(\bigvee C_1, \bigvee C_2)$ is the least upper bound of C.

- $\tau = \tau_1 \to \tau_2$

  Note $[\![\tau]\!] = \{f \mid [\![\tau_1]\!] \to [\![\tau_2]\!] : f \text{ is continuous}\}$. By inductive hypothesis, $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ are CPOs, and thus have bottom elements $\bot_1$ and $\bot_2$, respectively. Let $\bot$: $[\![\tau_1]\!] \to [\![\tau_2]\!]$ be the function such that, for all $x \in [\![\tau_1]\!],\ \bot\ (x) = \bot_2$. See, then, that $\bot$ is a continuous function that is less than or equal to every function in $[\![\tau]\!]$ (using pointwise ordering), so $\bot$ is the bottom element of $[\![\tau]\!]$.

  Let $C \subset [\![\tau]\!]$ be a chain, and consider that for all $x \in \tau_1, \{f(x) \mid f \in C\}$ is a chain in $[\![\tau_2]\!]$. Call this set F_x, and note that $\bigvee F_x$ exists. We define a function $g : [\![\tau_1]\!] \to [\![\tau_2]\!]$ such that $\forall x \in [\![\tau_1]\!],\ g(x) = \bigvee F_x$. We claim that $g = \bigvee C$.

  Let $f \in C$, and note that for all $x \in [\![\tau_1]\!],\ f(x) \leq \bigvee F_x = g(x)$. Then g is an upper bound of C.

Now let $\rho \in [\![\tau]\!]$ be an upper bound of C. Then for all $x \in [\![\tau_1]\!], f \in C,\ f(x) \leq \rho(x)$, so $\rho(x)$ is an upper bound of $\{f(x) : f \in C\} = F_x$. Thus $\rho(x) \geq F_x = g(x),\ \forall x \in [\![\tau_1]\!]$, so $g \leq \rho$. Therefore, g is the least upper bound of C.

□

Further, we must show that the argument $\lambda f.\lambda x.e$ of a `fix` expression interprets to a continuous function. It will suffice to prove type soundness of our denotational semantics: $\lambda f \lambda x.e$ expressions have type $(\tau \to \sigma) \to (\tau \to \sigma)$, and $[\![(\tau \to \sigma) \to (\tau \to \sigma)]\!]$ is the set of continuous functions from $[\![\tau \to \sigma]\!]$ to $[\![\tau \to \sigma]\!]$.

We will see, that in order to prove type soundness for $\lambda x.e$ expressions, we will need to show that $[\![\lambda x.e]\!]$ is a continuous function. Thus, the statement of type soundness has some additional complexities.

THEOREM 4. *For all expressions e and environments Γ,*

(1) For all chains $a_0 \leq a_1 \leq \cdots$,

$$[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\}.$$

(2) For all $\Gamma \vdash e : \tau$ and all Γ-environments η, $[\![\Gamma \vdash e : \tau]\!]\eta \in [\![\tau]\!]$.

PROOF. By induction on the structure of e. For all cases, let $\{a_i\}_{i=1}^{\infty}$ be a chain, and let $a = \bigvee_i a_i$. Note that 2 is trivial in all but the abstraction and `fix` cases.

Base Cases:

- $e \in \{\texttt{0},\ \texttt{1},\ \ldots\}$

- $e \in \{\texttt{true}, \texttt{false}\}$

  In both of these cases, $[\![\Gamma \vdash e : \tau]\!]$ is a constant function from a Γ-environment η to an element of $[\![\tau]\!]$, so clearly $[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\}$.

- $e = x : \tau$

  We want to show that $[\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto a_i\}$.

  In the case that $y \neq x$, note that

$$\begin{aligned}
[\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto \bigvee_i a_i\} &= \eta\{y \mapsto \bigvee_i a_i\}(x) \\
&= \eta(x) \\
&= \bigvee_i [\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto a_i\}
\end{aligned}$$

  In the case that $y = x$, note that

$$\begin{aligned}
[\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto \bigvee_i a_i\} &= \eta\{y \mapsto \bigvee_i a_i\}(x) \\
&= \bigvee_i a_i \\
&= \bigvee_i \eta\{y \mapsto a_i\}(x) \\
&= \bigvee_i [\![\Gamma \vdash x : \tau]\!]\eta\{y \mapsto a_i\}
\end{aligned}$$

Inductive Cases:

- $e = e_1 \circ e_2,\ \circ \in \{+, -, *, /\}$

  We assume that $\circ$ is continuous in each argument. Then

$$[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = [\![\Gamma \vdash e_1 : \tau]\!]\eta\ \{x \mapsto \bigvee_i a_i\} \circ [\![\Gamma \vdash e_2 : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\}$$

$$\text{(by inductive hypothesis)} = \bigvee_i [\![\Gamma \vdash e_1 : \tau]\!]\eta\{x \mapsto a_i\} \circ \bigvee_i [\![\Gamma \vdash e_2 : \tau]\!]\eta\{x \mapsto a_i\}$$

$$= \bigvee_i ([\![\Gamma \vdash e_1 : \tau]\!]\eta\{x \mapsto a_i\} \circ [\![\Gamma \vdash e_2 : \tau]\!]\eta\{x \mapsto a_i\})$$

$$= \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\}$$

- $e = e_1 \circ e_2,\ \circ \in \{=, <, >, \leq, \geq\}$

  We have three cases to consider: either $[\![\Gamma \vdash e_1 : \tau]\!]\{x \mapsto \bigvee_i a_i\} = \bot$, $[\![\Gamma \vdash e_2 : \tau]\!]\{x \mapsto \bigvee_i a_i\} = \bot$, or both $[\![\Gamma \vdash e_1 : \tau]\!]\{x \mapsto \bigvee_i a_i\} \neq \bot$ and $[\![\Gamma \vdash e_2 : \tau]\!]\{x \mapsto \bigvee_i a_i\} \neq \bot$.

  In the case that $[\![\Gamma \vdash e_1 : \tau]\!]\{x \mapsto \bigvee_i a_i\} = \bot$, by inductive hypothesis $\bigvee_i [\![\Gamma \vdash e_1]\!]\{x \mapsto a_i\} = \bot$, so $[\![\Gamma \vdash e_1]\!]\{x \mapsto a_i\} = \bot$ for all i. Then

$$[\![\Gamma \vdash e_1 \circ e_2]\!]\eta\{x \mapsto \bigvee_i a_i\} = [\![\Gamma \vdash e_1]\!]\eta\{x \mapsto \bigvee_i a_i\} \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto \bigvee_i a_i\}$$

$$= \bot \circ [\![e_2]\!]\{x \mapsto \bigvee_i a_i\} = \bot$$

  Also,

$$\bigvee_i [\![\Gamma \vdash e_1 \circ e_2]\!]\eta\{x \mapsto a_i\} = \bigvee_i \{[\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\}\}$$

$$= \bigvee_i \{\bot \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\}\}$$

$$= \bigvee_i \{\bot\} = \bot$$

  The same reasoning holds when $[\![\Gamma \vdash e_2 : \tau]\!]\{x \mapsto \bigvee_i a_i\} = \bot$.

In the case that both $[\![\Gamma \vdash e_1 : \tau]\!]\{x \mapsto \bigvee_i a_i\} \neq \bot$ and $[\![\Gamma \vdash e_2 : \tau]\!]\{x \mapsto \bigvee_i a_i\} \neq \bot$, by inductive hypothesis $\bigvee_i [\![\Gamma \vdash e_1 : \tau]\!]\{x \mapsto a_i\} \neq \bot$ and $\bigvee_i [\![\Gamma \vdash e_2 : \tau]\!]\{x \mapsto a_i\} \neq \bot$. Note $[\![\texttt{real}]\!] = \mathbb{R}_\bot$, so

$$\exists i_1 \forall i \geq i_1, [\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} = v_1 \in \mathbb{R}$$

$$\exists i_2 \forall i \geq i_2, [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\} = v_2 \in \mathbb{R}$$

Let $i' = \max\{i_1, i_2\}$, and note

$$\begin{aligned}
[\![\Gamma \vdash e_1 \circ e_2]\!]\eta\{x \mapsto \bigvee_i a_i\} &= [\![\Gamma \vdash e_1]\!]\eta\{x \mapsto \bigvee_i a_i\} \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto \bigvee_i a_i\} \\
\text{(by inductive hypothesis)} \quad &= \bigvee_i [\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} \circ \bigvee_i [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\} \\
&= \bigvee_{i \geq i'} [\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} \circ \bigvee_{i \geq i'} [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\} \\
&= v_1 \circ v_2
\end{aligned}$$

Also,

$$\begin{aligned}
\bigvee_i [\![\Gamma \vdash e_1 \circ e_2]\!]\eta\{x \mapsto a_i\} &= \bigvee_i \{[\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\}\} \\
&= \bigvee_{i \geq i'} \{[\![\Gamma \vdash e_1]\!]\eta\{x \mapsto a_i\} \circ [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto a_i\}\} \\
&= \bigvee_{i \geq i'} \{v_1 \circ v_2\} = v_1 \circ v_2
\end{aligned}$$

- $e =$ `if` e_1 `then` e_2 `else` $e_3 : \tau$

  In the case that $[\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bot$, note that $[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bot$. By inductive hypothesis, $[\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e_1 : \tau]\!]\eta\{x \mapsto a_i\} = \bot$. Then it must be that $\forall i, [\![\Gamma \vdash e_1 : \tau]\!]\eta\{x \mapsto a_i\} = \bot$. Thus, $\bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\} = \bot$, so $[\![\Gamma \vdash e :$

$\tau]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\}$.

In the case that $[\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto \bigvee_i a_i\} = \mathit{true}$, note that $[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\} = [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto \bigvee_i a_i\}$. By inductive hypothesis, $[\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e_1 : \tau]\!]\eta\{x \mapsto a_i\} = \mathit{true}$. Since $\{a_i\}_{i=1}^{\infty}$ is a chain and $[\![\texttt{bool}]\!]$ is a flat CPO, there must exist $n \in \mathbb{N}$ such that $\forall i > n, [\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto a_i\} = \mathit{true}$ and $\forall i \leq n, [\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto a_i\} = \bot$. Thus, it must be that $\bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\} = [\![\Gamma \vdash e_2]\!]\eta\{x \mapsto \bigvee_i a_i\} = [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\}$.

In the case that $[\![\Gamma \vdash e_1 : \texttt{bool}]\!]\eta\{x \mapsto \bigvee_i a_i\} = \mathit{false}$, by reasoning parallel to the previous case $\bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\} = [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\}$.

- $e = (e_1, e_2)$

$$[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\eta\{x \mapsto a\} = ([\![\Gamma \vdash e_1 : \tau_1]\!]\eta\{x \mapsto a\}, [\![\Gamma \vdash e_2 : \tau_2]\!]\eta\{x \mapsto a\})$$

$$\text{(by inductive hypothesis)} = (\bigvee_i [\![\Gamma \vdash e_1 : \tau_1]\!]\eta\{x \mapsto a_i\}, \bigvee_i [\![\Gamma \vdash e_2 : \tau_2]\!]\eta\{x \mapsto a_i\})$$

$$= \bigvee_i ([\![\Gamma \vdash e_1 : \tau_1]\!]\eta\{x \mapsto a_i\}, [\![\Gamma \vdash e_2 : \tau_2]\!]\eta\{x \mapsto a_i\})$$

$$= \bigvee_i [\![\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!]\eta\{x \mapsto a_i\}$$

- $e = \texttt{floor}(e_1)$

$$[\![\Gamma \vdash e]\!]\eta\{x \mapsto a\} = \lfloor [\![e_1]\!]\eta\{x \mapsto a\} \rfloor$$

$$\text{(by inductive hypothesis)} = \bigvee_i \lfloor [\![e_1]\!]\eta\{x \mapsto a_i\} \rfloor$$

- $e = e_1\ e_2$

We use Lemma 5 in this case, which is stated after the proof.

Let $\alpha = [\![\Gamma \vdash e_1 : \sigma \to \tau]\!]\eta$, $\beta = [\![\Gamma \vdash e_2 : \sigma]\!]\eta$. By inductive hypothesis,

$$\alpha\{x \mapsto \bigvee_i a_i\} = \bigvee_i \alpha\{x \mapsto a_i\} \text{ and}$$
$$\beta\{x \mapsto \bigvee_j a_j\} = \bigvee_j \beta\{x \mapsto a_j\}$$

For all i, $\alpha\{x \mapsto a_i\} \in [\![\sigma \to \tau]\!]$, which is the set of continuous functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$. Further, since $\{a_i\}_i$ is a chain, $\{\alpha\{x \mapsto a_i\}\}_i$ is a chain as well. Then, since $[\![\sigma \to \tau]\!]$ is a CPO, the supremum of $\{\alpha\{x \mapsto a_i\}\}_i$ is an element of $[\![\sigma \to \tau]\!]$. That is, $\bigvee \alpha\{x \mapsto a_i\}$ is a continuous function from $[\![\sigma]\!]$ to $[\![\tau]\!]$. Then

$$\begin{aligned}
[\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto \bigvee_j a_j\} &= \alpha\{x \mapsto \bigvee_i a_i\}(\beta\{x \mapsto \bigvee_j a_j\}) \\
&= (\bigvee_i \alpha\{x \mapsto a_i\})(\bigvee_j \beta\{x \mapsto a_j\}) \\
\text{(by continuity of } \bigvee_i \alpha\{x \mapsto a_i\}) &= \bigvee_j (\bigvee_i \alpha\{x \mapsto a_i\})(\beta\{x \mapsto a_j\}) \\
\text{(by Lemma 5)} &= \bigvee_i \alpha\{x \mapsto a_i\}(\beta\{x \mapsto a_i\}) \\
&= \bigvee_i [\![\Gamma \vdash e : \tau]\!]\eta\{x \mapsto a_i\}
\end{aligned}$$

- $e = \lambda(y : \tau).(e' : \tau')$
  (1) Then $[\![\Gamma \vdash e : \tau \to \tau']\!]\eta\{x \mapsto \bigvee_i a_i\}$ is a function $f : [\![\tau]\!] \to [\![\tau']\!]$ such that $\forall d \in [\![\tau]\!]$, $f(d) = [\![\Gamma\ y : \tau \vdash e' : \tau]\!]\eta\{x \mapsto \bigvee_i a_i\}\{y \mapsto d\}$. We want to show that

$$[\![\Gamma \vdash e : \tau \to \tau']\!]\eta\{x \mapsto \bigvee_i a_i\} = \bigvee_i [\![\Gamma \vdash e : \tau \to \tau']\!]\eta\{x \mapsto a_i\}$$

Let $d \in [\![\tau]\!]$, and consider that

$$\begin{aligned}
[\![\Gamma \vdash e : \tau \to \tau']\!]\eta\{x \mapsto \bigvee_i a_i\}(d) &= [\![\Gamma.y : \tau \vdash e' : \tau']\!]\eta\{x \mapsto \bigvee_i a_i\}\{y \mapsto d\} \\
&= [\![\Gamma.y : \tau \vdash e' : \tau']\!](\eta\{y \mapsto d\})\{x \mapsto \bigvee_i a_i\} \\
\text{(by inductive hypothesis)} &= \bigvee_i [\![\Gamma.y : \tau \vdash e' : \tau']\!](\eta\{y \mapsto d\})\{x \mapsto a_i\} \\
&= \bigvee_i [\![\Gamma.y : \tau \vdash e' : \tau']\!](\eta\{x \mapsto a_i\})\{y \mapsto d\} \\
&= \bigvee_i [\![\Gamma \vdash \lambda y : \tau.e' : \tau']\!]\eta\{x \mapsto a_i\}(d) \\
&= \bigvee_i [\![\Gamma \vdash e : \tau \to \tau']\!]\eta\{x \mapsto a_i\}(d)
\end{aligned}$$

(2) We want to show that $[\![\Gamma \vdash e : \tau \to \tau']\!]\eta \in [\![\tau \to \tau']\!]$—that is, it is a continuous function from $[\![\tau]\!]$ to $[\![\tau']\!]$. Let $\{a_i\}_{i=1}^{\infty}$ be a chain of elements in $[\![\tau]\!]$, and see that

$$\begin{aligned}
[\![\Gamma \vdash e : \tau \to \tau']\!]\eta(\bigvee_i a_i) &= [\![\Gamma.x : \tau \vdash e' : \tau']\!]\eta\{x \mapsto \bigvee_i a_i\} \\
\text{(by inductive hypothesis)} &= \bigvee_i [\![\Gamma.x : \tau \vdash e' : \tau']\!]\eta\{x \mapsto a_i\} \\
&= \bigvee_i [\![\Gamma \vdash e : \tau \to \tau']\!]\eta(a_i)
\end{aligned}$$

- $e = \texttt{fix}(\lambda f.\lambda y.e)$

  We use several Lemmas 6 and 7 for this case, which are stated after the proof.

(1) Let $F_i = [\![\Gamma \vdash \lambda f.\lambda y.e]\!]\eta\{x \mapsto a_i\}$, and consider that

$$\begin{aligned}
[\![\texttt{fix}(\lambda f.\lambda y.e)]\!]\eta\{x \mapsto \bigvee_i a_i\} &= fix([\![\lambda f.\lambda y.e]\!]\eta\{x \mapsto \bigvee_i a_i\}) \\
\text{(by inductive hypothesis)} &= fix(\bigvee_i (F_i)) \\
&= \bigvee_j (\bigvee_i ((F_i)^j)\ \bot) \\
&= \bigvee_j (\bigvee_i ((F_i)^j\ \bot)) \\
\text{(by Lemma 6)} &= \bigvee_i (\bigvee_j ((F_i)^j\ \bot)) \\
&= \bigvee_i (fix(F_i)) \\
&= \bigvee_i [\![\Gamma \vdash \texttt{fix}(\lambda f \lambda y.e)]\!]\eta\{x \mapsto a_i\}
\end{aligned}$$

(2) Let $F = [\![\Gamma \vdash \lambda f.\lambda x.e]\!]\eta$. We want to show that

$$[\![\Gamma \vdash \texttt{fix}(\lambda f \lambda x.e) : \sigma \to \tau]\!]\eta \in [\![\sigma \to \tau]\!].$$

That is, it is a continuous function from $[\![\sigma]\!]$ to $[\![\tau]\!]$. Let $\{a_i\}_{i=1}^{\infty}$ be a chain of elements in $[\![\sigma]\!]$, and see that

$$\begin{aligned}
([\![\Gamma \vdash \texttt{fix}(\lambda f \lambda x.e) : \tau \to \tau]\!]\eta)(\bigvee_i a_i) &= fix(F)(\bigvee_i a_i) \\
&= (\bigvee_j F^j \perp)(\bigvee_i a_i) \\
&= \bigvee_j (F^j \perp (\bigvee_i a_i)) \\
\text{(by continuity of } F^j \perp) \ &= \bigvee_j (\bigvee_i (F^j \perp (a_i))) \\
\text{(by Lemma 7)} &= \bigvee_i (\bigvee_j (F^j \perp (a_i))) \\
&= \bigvee_i (fix(F)(a_i)) \\
&= \bigvee_i ([\![\Gamma \vdash \texttt{fix}(\lambda f \lambda x.e) : \tau \to \tau]\!]\eta(a_i))
\end{aligned}$$

Thus, by definition, $[\![\Gamma \vdash \texttt{fix}(\lambda f \lambda x.e)]\!]\eta$ is a continuous function from $[\![\tau]\!]$ to $[\![\tau]\!]$. □

LEMMA 5. *Let $\{f_i\}_i$ be a chain of functions of type $[\![\sigma]\!] \to [\![\tau]\!]$, and let $\{x_i\}_i$ be a chain in $[\![\sigma]\!]$. Then*

$$\bigvee_j \{(\bigvee_i f_i)(x_j)\} = \bigvee_i \{f_i(x_i)\}$$

PROOF. $\Rightarrow$ Towards contradiction, suppose $\bigvee_j\{(\bigvee_i f_i)(x_j)\} < \bigvee_i\{f_i(x_i)\}$. Then

$$\exists k \text{ s.t. } \bigvee_j\{(\bigvee_i f_i)(x_j)\} < f_k(x_k)$$

$$\Rightarrow (\bigvee_i f_i)(x_k) < \bigvee_j\{(\bigvee_i f_i)(x_j)\} < f_k(x_k) \Rightarrow\Leftarrow$$

Therefore, $\bigvee_j\{(\bigvee_i f_i)(x_j)\} \geq \bigvee_i\{f_i(x_i)\}$

$\Leftarrow$ Suppose, now, that $\bigvee_j\{(\bigvee_i f_i)(x_j)\} > \bigvee_i\{f_i(x_i)\}$. Then

$$\exists k_1 \text{ s.t. } (\bigvee_i f_i)(x_{k_1}) > \bigvee_i\{f_i(x_i)\}$$

$$\Rightarrow \exists k_1, k_2 \text{ s.t. } f_{k_2}(x_{k_1}) > \bigvee_i\{f_i(x_i)\}$$

Let $k = \max(k_1, k_2)$. Then

$$f_{k_1}(x_{k_2}) > \bigvee_i\{f_i(x_i)\} > f_k(x_k) \Rightarrow\Leftarrow$$

Therefore, $\bigvee_j\{(\bigvee_i f_i)(x_j)\} \leq \bigvee_i\{f_i(x_i)\}$, so $\bigvee_j\{(\bigvee_i f_i)(x_j)\} = \bigvee_i\{f_i(x_i)\}$ $\square$

LEMMA 6. $\bigvee_j(\bigvee_i((F_i)^j\ \bot)) = \bigvee_i(\bigvee_j((F_i)^j\ \bot))$

PROOF. $\Rightarrow$ We want to show that

$$\bigvee_j(\bigvee_i((F_i)^j\ \bot)) \geq \bigvee_i(\bigvee_j((F_i)^j\ \bot))$$

First, notice that $\forall i, j\ \bigvee_j(\bigvee_i((F_i)^j\ \bot)) \geq (F_i)^j\ \bot$.

Suppose, now, that

$$\begin{aligned} & \bigvee_j (\bigvee_i ((F_i)^j\ \bot)) < \bigvee_i (\bigvee_j ((F_i)^j\ \bot)) \\ \Rightarrow \exists i \text{ s.t.} \quad & \bigvee_j (\bigvee_i ((F_i)^j\ \bot)) < \bigvee_j (F_i)^j \\ \Rightarrow \exists i, j \text{ s.t.} \quad & \bigvee_j (\bigvee_i ((F_i)^j\ \bot)) < (F_i)^j\ \bot \ \ \Rightarrow\Leftarrow \end{aligned}$$

Thus, it must be that $\bigvee_j(\bigvee_i((F_i)^j\ \bot)) \geq \bigvee_i(\bigvee_j((F_i)^j\ \bot))$

$\Leftarrow$ By parallel reasoning, $\bigvee_j(\bigvee_i((F_i)^j\ \bot)) \leq \bigvee_i(\bigvee_j((F_i)^j\ \bot))$

Therefore, $\bigvee_j(\bigvee_i((F_i)^j\ \bot)) = \bigvee_i(\bigvee_j((F_i)^j\ \bot))$ □

LEMMA 7. $\bigvee_j(\bigvee_i(F^j\ \bot\ (x_i))) = \bigvee_i(\bigvee_j(F^j\ \bot\ (x_i)))$

PROOF. By reasoning similar to Lemma 6. □

We will prove equational soundness of our denotational semantics, in order to ensure that our interpretation of recurrences reflects their equational meaning.

THEOREM 8. *If* $\Gamma \vdash e_0 : \tau$, $\Gamma \vdash e_1 : \tau$, *and* $e_0 = e_1$, *then for all* Γ*-environments* η,

$$[\![\Gamma \vdash e_0]\!]\eta = [\![\Gamma \vdash e_1]\!]\eta$$

PROOF. We will verify only the equation for `fix` expressions, as all others are trivial.

We may show by a simple inductive argument that $[\![[x \mapsto e']e]\!]\eta = [\![e]\!]\eta\{x \mapsto [\![e']\!]\eta\}$. Let $\mathtt{F} = \lambda f \lambda x.e$, and see that

$$\begin{aligned}
[\![[f \mapsto \mathtt{fix(F)}]\lambda x.e]\!]\eta &= [\![\lambda x.e]\!]\eta\{f \mapsto [\![\mathtt{fix(F)}]\!]\} \\
&= [\![\lambda x.e]\!]\eta\{f \mapsto fix\}([\![\mathtt{F}]\!]\eta)\} \\
&= [\![\lambda f.\lambda x.e]\!]\eta(fix([\![\mathtt{F}]\!]\eta)\} \\
&= [\![F]\!]\eta(fix([\![\mathtt{F}]\!]\eta)) \\
&= fix([\![\mathtt{F}]\!]\eta) = [\![\mathtt{fix}(\lambda f.\lambda x.e)]\!]\eta
\end{aligned}$$

□

We will look at a recurrence Karp offers in this section as an example, to check whether recurrences in our language have the same solution. First, consider the following recurrence:

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 1 + T(pn),\ 0 < p < 1
\end{aligned}$$

Again, we assume that the actual meaning of the general recurrence is $T(n) = 1 + T(\lfloor pn \rfloor)$. We may express this recurrence in our language as follows:

$$\mathtt{T} = \mathtt{fix}(\lambda f.\lambda x.\mathtt{if}\ \ x = 1\ \ \mathtt{then}\ \ 0\ \ \mathtt{else}\ \ 1 + f(\mathtt{floor}(p * x)))$$

We want to show that the interpretation of this expression is equal to the solution of the recurrence described above. First, let

$$\mathtt{F} = \lambda f.\lambda x.\mathtt{if}\ \ x = 1\ \ \mathtt{then}\ \ 0\ \ \mathtt{else}\ \ 1 + f(\mathtt{floor}(p * x)),$$

and note the following:

$$[\![\texttt{T}]\!] = fix([\![\texttt{F}]\!])$$

$$[\![\texttt{F}]\!] = F : [\![\texttt{real} \to \texttt{real}]\!] \to [\![\texttt{real} \to \texttt{real}]\!]$$

$$\text{s.t. } \forall \alpha \in [\![\texttt{real}]\!] \to [\![\texttt{real}]\!],$$

$$F(\alpha) = [\![\lambda x.\texttt{if } x = 1 \texttt{ then } 0 \texttt{ else } 1 + f(\texttt{floor}(p * x))]\!]\{f \mapsto \alpha\}$$

$$[\![\texttt{real}]\!] = \mathbb{R}_\perp$$

By the CPO fixed-point theorem,

$$fix(F) = \bigvee_n (F^n(\perp))$$

$$= \lim_{n\to\infty} \{F^n(\perp)\}$$

LEMMA 9. $\forall x \in \mathbb{R}_\perp,\ F^{n+1}(\perp)(x) = \begin{cases} 0 \ \textit{if } \lfloor x \rfloor = 1 \\ 1 \ \textit{if } \lfloor px \rfloor = 1 \\ \dots \\ n \ \textit{if } \lfloor p^n x \rfloor = 1 \\ \perp \ \ \textit{otherwise} \end{cases}$

PROOF. *Base Case:* $n = 0$

$$F^{n+1}(\bot)(x) = F(\bot)$$

$$= \begin{cases} 0 \text{ if } \lfloor x \rfloor = 1 \\ 1 + \bot\ (\lfloor px \rfloor) \text{ otherwise} \end{cases}$$

$$= \begin{cases} 0 \text{ if } \lfloor x \rfloor = 1 \\ \bot \text{ otherwise} \end{cases}$$

Note $p^n x = p^0 x = x$, so trivially the claim holds.

Inductive Case: Suppose the claim holds for all $k < n$.

$$F^{n+1}(\bot)(x) = \begin{cases} 0 \text{ if } \lfloor x \rfloor = 1 \\ 1 + F^n(\bot)(px) \text{ otherwise} \end{cases}$$

By inductive hypothesis,

$$F^n(\bot)(px) = F^{(n-1)+1}(\bot)(px)$$

$$= \begin{cases} 0 \text{ if } \lfloor px \rfloor = 1 \\ 1 \text{ if } \lfloor p^2 x \rfloor = 1 \\ \dots \\ n-1 \text{ if } \lfloor p^n x \rfloor = 1 \\ \bot \text{ otherwise} \end{cases}$$

□

See, then, that

$$F^{n+1}(\bot)(x) = \begin{cases} 0 \text{ if } \lfloor x \rfloor = 1 \\ 1 \text{ if } \lfloor px \rfloor = 1 \\ 2 \text{ if } \lfloor p^2 x \rfloor = 1 \\ \dots \\ n \text{ if } \lfloor p^n x \rfloor = 1 \\ \bot \text{ otherwise} \end{cases}$$

Therefore, the claim holds for all n.

Then $[\![\mathtt{T}]\!] = \lim_{n\to\infty}\{F^n(\bot)\}$ is defined for all $x \in \mathbb{R}$ as

$$\lim_{n\to\infty}\{F^n(\bot)\}(x) = k$$

Where $\lfloor p^k x \rfloor = 1$. Then $k = \lfloor \log_p 1/x \rfloor$, so $[\![\mathtt{T}]\!](x) = \lfloor \log_p 1/x \rfloor = T(x)$. Thus, $[\![\mathtt{T}]\!] = T$.

CHAPTER 4

Reconciling the Two Semantics

Chapters 2 and 3 have presented languages for defining the two distinct kinds of deterministic recurrences present in Karp [1994]. In this chapter, we look at several attempts to connect these two recurrences. In particular, we want to find a way to write a recurrence with an initial condition as a recurrence of the form $T(x) = a(x) + T(m(x))$, as this is the kind of recurrence for which Karp proves tail bounds. We will offer several methods of translating from one expression language to the other, and investigate the validity of these approaches. Our goal here is to show that, for every expression of the form $\texttt{fix}(\lambda f.\lambda x.e)$ in our chapter 3 language, there are a, m in our chapter 2 language such that $[\![\texttt{rec}(a, m)]\!]_{sum} = [\![\texttt{fix}(\lambda f.\lambda x.e)]\!]_{fix}$. We use $[\![\cdot]\!]_{sum}$ and $[\![\cdot]\!]_{rec}$ to denote the semantics of chapters 2 and 3, respectively. We will see that this will only be possible for e of a certain form, but that this form covers all the examples that Karp provides.

As a first attempt, we consider a simple recurrence with an initial condition, as described in chapter 3.

$$T(1) = 0$$

$$T(n) = 1 + T(n/2)$$

As in the example in chapter 3, we assume that the actual meaning of the general recurrence is $T(n) = 1 + T(\lfloor n/2 \rfloor)$. We can write this recurrence as an expression

$\mathtt{T}$ in our fixed-point language, where

$$[\![\mathtt{T}]\!]_{fix}(n) = \lfloor \log_2(n) \rfloor.$$

In order to make this fit the definition of a recurrence described in chapter 2 — that is, a recurrence of the form $T(n) = a(n) + T(m(n))$— we may interpret a as a piecewise function with two sub-functions: one for the general recurrence and one for the initial condition. We write this as a single recurrence

$$T'(n) = a(n) + T'(m(n))$$

$$m(n) = \lfloor n/2 \rfloor, \ \forall n$$

$$a(n) = \begin{cases} 0, \ n \leq 1 \\ 1, \ n > 1 \end{cases}$$

This recurrence can then be expressed in our infinite sum language as

$$\mathtt{T} = \mathtt{rec}(a, m)$$

$$m = \lambda x.\mathtt{floor}(x/2)$$

$$a = \lambda x.\mathtt{if} \ \ x \leq 1 \ \ \mathtt{then} \ 0 \ \mathtt{else} \ 1$$

$$\begin{aligned} [\![\mathtt{T}]\!](n) &= [\![\mathtt{rec}(a, m)]\!](n) \\ &= \sum_{i=0}^{\infty} [\![a]\!]([\![m]\!]^i(n)) \\ &= \sum_{i=0}^{\lfloor \log_2(n) \rfloor} 1 + \sum_{i=\lfloor \log(n) \rfloor + 1}^{\infty} 0 \\ &= \lfloor \log_2(n) \rfloor. \end{aligned}$$

However, this approach will not work in every case. Let us try to use the same process on the following recurrence

$$T(1) = 1$$

$$T(n) = 2 + T(\lfloor n/2 \rfloor)$$

We first write this as a single recurrence. See that, unlike the previous example, a never maps to 0.

$$T'(n) = a(n) + T'(m(n))$$

$$m(n) = \lfloor n/2 \rfloor,\ \forall n$$

$$a(n) = \begin{cases} 1,\ n \leq 1 \\ 2,\ n > 1 \end{cases}$$

We then express this recurrence using our infinite sum language as

$$\mathtt{T} = \mathtt{rec}(a, m)$$

$$m = \lambda x.\mathtt{floor}(x/2)$$

$$a = \lambda x.\mathtt{if}\ \ x \leq 1\ \mathtt{then}\ 1\ \mathtt{else}\ 2$$

$$\begin{aligned} [\![\mathtt{T}]\!](d) &= [\![\mathtt{rec}(a, m)]\!](d) \\ &= \sum_{i=0}^{\infty} [\![a]\!]([\![m]\!]^i(d)) \\ &= \sum_{i=0}^{\lfloor \log(n) \rfloor} 2 + \sum_{i=\lfloor \log(n) \rfloor + 1}^{\infty} 1 \\ &= \infty \end{aligned}$$

Which, clearly, is not the same as the fixed-point semantics for T.

A second attempt takes a similar but slightly altered approach. Suppose that we interpret a as a piecewise function with three sub-functions: one for the general recurrence, one for the initial condition, and one for values smaller than the initial condition, which all map to 0. Considering the example that failed in the previous attempt, we would have

$$T'(n) = a(n) + T'(m(n))$$

$$m = \lfloor n/2 \rfloor, \forall n$$

$$a(n) = \begin{cases} 0, & n < 1 \\ 1, & n = 1 \\ 2, & n > 1 \end{cases}$$

We can express this recurrence using our infinite sum language as

$$\mathtt{T} = \mathtt{rec}(a, m)$$

$$m = \lambda x.\mathtt{floor}(x/2)$$

$$a = \lambda x.\mathtt{if}\ x < 1\ \mathtt{then}\ 0\ \mathtt{else}\ (\mathtt{if}\ x = 1\ \mathtt{then}\ 1\ \mathtt{else}\ 2).$$

See, then, that

$$[\![a]\!](n) = \begin{cases} 0, & n < 1 \\ 1, & n = 1 \\ 2, & n > 1 \end{cases}$$

$$\begin{aligned}
[\![\texttt{T}]\!](n) &= [\![\texttt{rec}(a, m)]\!](n) \\
&= \sum_{i=0}^{\infty} [\![a]\!]([\![m]\!]^i(n)) \\
&= \sum_{i=0}^{\lfloor \log(n) \rfloor} 2 + \sum_{i=\lfloor \log(n) \rfloor + 1}^{\lfloor \log(n) \rfloor + 1} 1 + \sum_{i=\lfloor \log(n) \rfloor + 2}^{\infty} 0 \\
&= \lfloor 2 \log(n) \rfloor + 1
\end{aligned}$$

This approach matches our intuition for how an initial condition should work—that once you reach the initial condition, you stop recursing.

With this in mind, we can identify a general strategy for extracting a recurrence in our chapter 2 syntax (with the `rec` operator), given a recurrence in our chapter 3 syntax (with the `fix` operator). What we would like to now show is that for any $\texttt{fix}(\lambda f.\lambda x.e)$ expression, there are expressions a and m in our infinite sum language such that $[\![\texttt{fix}(\lambda f.\lambda x.e)]\!]_{fix} = [\![\texttt{rec}(a, m)]\!]_{sum}$. But we cannot prove this exact theorem, as not every expression e is of a form that corresponds to a valid recurrence with an initial condition.

What we can do instead is define a sublanguage of our fixed-point language such that, for any expression E in this sublanguage, we may translate E to an expression E^* in our infinite sum language. We may then prove that for any expression E, $[\![E]\!]_{fix}$ is equivalent to $[\![E^*]\!]$. However, there are some underlying complexities in our semantics that make this proof of equivalence non-trivial; formalizing this translation and proof is left for future work.

Still, we can look at an example of what it might look like to make a translation from a `fix` expression corresponding to a chapter 3 recurrence to a `rec` expression corresponding to a chapter 2 recurrence. Suppose we have a recurrence with an

initial condition, given as follows:

$$T(b) = c, \text{ where } b, c \in \mathbb{R}, b > 0$$

$$T(x) = a(x) + T(m(x)).$$

We write this recurrence in our chapter 3 syntax as

$$\texttt{fix}(\lambda f.\lambda x.\texttt{if } x = b \texttt{ then } c \texttt{ else } a(x) + f((m(x)))),$$

an expression whose denotation will be a function F such that

$$F(x) = \begin{cases} c \text{ if } x = b \\ a(x) + c \text{ if } m(x) = b \\ a(x) + a(m(x)) + c \text{ if } m(m(x)) = b \\ \dots \\ \sum_{i=1}^{n} a(m^{i-1}(x)) + c \text{ if } m^n(x) = b \\ \dots \end{cases}$$

We may then write this recurrence in our chapter 2 syntax as $\texttt{rec}(a', m)$, where m is the same function as in the previous recurrence, and

$$a' = \lambda x.\lambda f.\texttt{if } x < b \texttt{ then } 0 \texttt{ else } (\texttt{if } x = b \texttt{ then } c \texttt{ else } a(x)).$$

Then the denotation of $\texttt{rec}(a', m)$ is a function F' such that for all x in the domain of F,

$$\begin{aligned} F'(x) &= \sum_{i=0}^{\infty} [\![a']\!]([\![m]\!]^i(x)) \\ &= \sum_{i=0}^{n-1} [\![a']\!]([\![m]\!]^i(x)) + \sum_{i=n}^{n} c + \sum_{i=n+1}^{\infty} 0 \\ &\quad \text{where } [\![m]\!]^n(x) = b \\ &= \sum_{i=1}^{n} [\![a']\!]([\![m]\!]^{i-1}(x)) + c \end{aligned}$$

See, then, that $F' = F$. Therefore, this confirms that, given an expression e of the specific form $e = \texttt{if } x = m \texttt{ then } c \texttt{ else } a(x) + f(m(x))$, we can extract a function a' such that

$$[\![\texttt{fix}(\lambda f.\lambda x.e)]\!]_{fix} = [\![\texttt{rec}(a', m)]\!]_{sum}$$

CHAPTER 5

Well-Typed Probabilistic Recurrence Relations

We now turn to recurrences describing the cost of probabilistic algorithms. Karp [1994] is premised on the notion that the cost of a stochastic algorithm may be described as a recurrence relation of the form

$$T(x) = a(x) + T(h_1(x)) + \cdots + T(h_n(x)).$$

But what does this equation actually mean? To unpack this, we will attempt to assign types to this recurrence.

Following Karp's definitions, we can immediately assign types to some of these terms:

$T(x) : \Omega_x \to \mathbb{R}$, where Ω_x is the sample space of all problem instances of size x

$$T : \prod_{x \in \mathbb{R}} (\Omega_x \to \mathbb{R})$$

$$a : \mathbb{R} \to \mathbb{R}$$

Notice that the type of T is dependent upon a real number x. For other terms, it is less clear what the type should be. Karp alternates between saying that h is a random variable and $h(x)$ is a random variable, so it is not immediately obvious what the meaning of this function is. However, since h takes $x \in \mathbb{R}$ as an input, it would not make sense for h to be a random variable of type $\mathbb{R} \to \mathbb{R}$. This would make h a function that takes an input size and returns the size of the derived subproblem—thus sidestepping the stochastic part of the algorithm. Then

it must be that $h(x)$ is a random variable on a specific input of size x—like $T(x)$, it takes an element of the sample space Ω_x and returns the size of the derived subproblem.

$$h_i(x) : \Omega_x \to \mathbb{R}$$

$$h_i : \prod_{x \in \mathbb{R}} (\Omega_x \to \mathbb{R})$$

However, there are some inconsistencies in these type assignments. To start, T is a function that takes a real number and returns a random variable. But on the right-hand side of the relation, T takes $h_i(x)$ as an argument; $h_i(x)$ is not a real number, but a random variable.

Thinking in terms of what we *want* this recurrence to express, the arguments to the T's on the righthand side of the equation should be the sizes of the derived subproblems—that is, the result of applying $h_i(x)$ to the original input. Then, given an input $l \in \Omega_x$, we should have

$$T(x)(l) = a(x) + T(h_1(x)(l)) \cdots + T(h_n(x)(l))$$

$$\text{where } h(x)(l) : \mathbb{R}$$

Thus, the term $T(h_i(x)(l))$ typechecks. However, we now run into another problem: $T(x)(l)$ and $a(x)$ are real numbers, but $T(h_i(x)(l))$ is a random variable of type $\Omega_{h_i(x)(l)} \to \mathbb{R}$. Then $T(h_i(x)(l))$ needs an argument. Consider that $T(h_i(x)(l))$ is a random variable describing the running time of the algorithm with an input of size $h_i(x)(l)$. For instance, given a recurrence for randomized quicksort, $T(h_1(x)(l))$ and $T(h_2(x)(l))$ describe how long it takes to quicksort the left and right sublists of an input list l, which themselves have length $h_1(x)(l)$ and

$h_2(x)(l)$. The arguments to these random variables, then, should be the left and right sublists themselves.

In order for us to obtain these sublists, it will be necessary for us to define a function

$$\hat{h_i}(x) : \prod_{l \in \Omega_x} \Omega_{h_i(x)(l)}$$

$$\hat{h_i} : \prod_{x \in R} (\prod_{l \in \Omega_x} \Omega_{h_i(x)(l)})$$

such that $\hat{h_i}(x)$ takes an input of size x and returns the derived subproblem of size $h_i(x)(l)$. See, then, that $T(h_i(x)(l))(\hat{h_i}(x)(l))$ has type $\mathbb{R}$. So, this leaves us with the equation

$$T(x)(l) = a(x) + T(h_1(x)(l))(\hat{h_1}(x)(l)) \cdots + T(h_n(x)(l))(\hat{h_n}(x)(l))$$

which, finally, typechecks.

Thus, we find that attempting to formally assigns types to Karp's probabilistic recurrences reveals a lot of hidden complications and ambiguities, which Karp never addresses directly. Most notably, we find that the types of the functions T, h, and $\hat{h}$ are dependent upon a real number x, describing the size of problems in the sample space.

We define a language to write and type-check these recurrences. This language is based on λLF—a simple system of dependent types (Pierce [2005]). However, instead of general dependent types, our language only needs to describe sample space types Ω_n which are dependent upon a natural number n. Thus, we add a constant `list` to our syntax, which works as a function from expressions to types. That is, for each expression e of type `nat`, `list`(e) is a type. There are many stochastic divide-and-conquer algorithms involving sample spaces on datatypes

other than lists; this language could easily be expanded to include other constants such as `tree`, `graph`, etc. Note that lists can only be indexed over natural numbers (it would not make sense for a list to be any other size), so we include the type `nat` in our language as well.

$$\begin{aligned}
\tau &::= \texttt{real} \mid \texttt{nat} \mid \texttt{bool} \mid \tau \times \tau \mid \Pi x : \tau.\tau \mid \texttt{list}(e) \\
e &::= x \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \lambda x.e \mid e\ e \mid e + e \mid e - e \mid e * e \mid e/e \mid \texttt{true} \mid \texttt{false} \mid \\
&\quad e = e \mid e < e \mid e > e \mid e \leq e \mid e \geq e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid (e, e) \mid \\
&\quad \texttt{rec}(a, (h_1, \ldots, h_n), (\hat{h_1}, \ldots \hat{h_n}))
\end{aligned}$$

In this syntax, we replace the arrow type $\tau \to \sigma$ with the dependent product type $\Pi(x : \tau).\sigma$, and introduce type families. These are collections of types that depend on an input: for instance, the type $\Pi n : \texttt{nat}.\texttt{list}(n)$, where the type $\texttt{list}(n)$ is dependent upon an input n of type `nat`. As with our languages for deterministic recurrences, we need a way of defining recursive functions.

We could attempt to define a PCF-like language analogous to the one in chapter 3, with a `fix` operator that assigns the least fixed-point to continuous functions. However, the inclusion of dependent product types creates some unique problems in our denotational semantics. Consider that in standard PCF, the denotation of the arrow type, $[\![\tau \to \sigma]\!]$, is the set of continuous functions of type $[\![\tau]\!] \to [\![\sigma]\!]$. But for dependent product types $\Pi(x : \tau).\sigma$, this denotation does not make sense: since σ may be dependent upon expressions of type τ, there is no guarantee that σ is a well-formed type (for instance, given the type $\Pi(n : \texttt{nat}).\texttt{list}(n)$, see that $\texttt{list}(n)$ is not well-formed because we do not know what n is).

Thus, it makes sense to alter the denotation of a dependent product type $\Pi(x : \tau).\sigma$ to instead be the set of all continuous functions f of type $[\![\tau]\!] \to \bigcup_{a \in [\![\tau]\!]} [\![\sigma]\!]\{x \mapsto a\}$ where $\forall a \in [\![\tau]\!],\ f(a) \in [\![\sigma]\!]\{x \mapsto a\}$. If we again look at the

$$\Gamma \vdash \texttt{real} :: *$$

$$\Gamma \vdash \texttt{bool} :: *$$

$$\Gamma \vdash \texttt{nat} :: *$$

$$\frac{\Gamma \vdash \tau_1 :: * \quad \Gamma, x : \tau_1 \vdash \tau_2 :: *}{\Gamma \vdash \Pi x : \tau_1 . \tau_2 :: *}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{list}(e) :: *}$$

FIGURE 1. Well-formed types for dependent type language.

type $\Pi(n : \texttt{nat}).\texttt{list}(n)$, we see that its denotation is the set of all functions that map a natural number n to a list of length n.

But there is a problem here, stemming from the fact that our denotation of dependent product types must be a set of *continuous* functions. This is the case because the denotation of our fix operator, $[\![\texttt{fix}(\lambda f.\lambda x.e)]\!]$, is $fix([\![\lambda f.\lambda x.e]\!])$, where fix is a function that assigns the least fixed point to continuous functions. Thus, we must be able to guarantee that the denotation of any $\lambda f.\lambda x.e$ expression is a continuous function. But continuity of a function is premised on the fact that the domain and range of a function are CPOs. It is not necessarily the case that $\bigcup_{a \in [\![\tau]\!]} [\![\sigma]\!]\{x \mapsto a\}$ is a CPO, even though $[\![\sigma]\!]\{x \mapsto a\}$ is a CPO for all $a \in [\![\tau]\!]$. For instance, $[\![\texttt{list}(n)]\!]\{n \mapsto a\}$ is a CPO for all $a \in \mathbb{N}$, but $\bigcup_{a \in \mathbb{N}} [\![\texttt{list}(n)]\!]\{n \mapsto a\}$—the union of all samples spaces Ω_n—is not.

As a result, continuity does not have any meaning for functions of type $[\![\Pi(x : \tau).\sigma]\!]$. So we cannot use the `fix` operator on arbitrary $\lambda f.\lambda x.e$ expressions, and the whole framework of our denotational semantics falls apart.

As an alternative approach, we define a language which includes a `rec` operator similar to the one in chapter 2, which allows to write recurrences of the form

$T(x)(l) = a(x) + T(h_1(x)(l))\hat{h_1}(x)(l) + \ldots + T(h_n(x)(l))\hat{h_n}(x)(l)$, where $T,\ a,\ h_i$, and $\hat{h_i}$ are functions with types as described above. As in chapter 2, this approach restricts us to expressing only a subset of recursive functions; however, it does allow us to express the probabilistic recurrences described by Karp, without having to worry about the problem of requiring continuity on functions with dependent product types.

We offer standard type judgements in Figure 2, as well as judgements for when a type is well-formed, denoted by $\tau :: *$, in Figure 1. We also offer an equational semantics in Figure 3 and denotational semantics for types and expressions in Figures 4 and 5. We may verify soundness of our denotational semantics with an argument analogous to the one given in chapter 2.

THEOREM 10. *If* $\Gamma \vdash e_0 : \tau$ *and* $\Gamma \vdash e_1 : \tau$ *and* $e_0 = e_1$*, then for all* Γ*-environments* η*,*

$$[\![\Gamma \vdash e_0]\!]\eta = [\![\Gamma \vdash e_1]\!]\eta$$

We want to gain some intuition that our denotation of a `rec` expression is equivalent to the solution of the corresponding semantic recurrence. To this end, we will show that, for a program T describing a recurrence for quicksort, the denotation of T is equal to the actual cost of quicksorting. Following Karp's definition, this random variable has the form

$$T(x)(l) = a(x) + T(h_1(x)(l))(\hat{h_1}(x)(l)) + T(h_2(x)(l))(\hat{h_2}(x)(l)),$$

where $a(x) = x - 1$, the cost of one step of quicksort (i.e. comparing all other elements in the list to the head of the list), $h_1(x)$ and $h_2(x)$ give the sizes of the left and right sublists of an input, and $\hat{h_1}$ and $\hat{h_2}$ give the sublists themselves.

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{real}}, \quad \frac{}{\Gamma \vdash \texttt{1} : \texttt{real}}, \ldots$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}, \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{x : \tau \in \Gamma \quad \Gamma \vdash \tau :: *}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash \sigma :: * \quad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x : \sigma).e : \Pi x : \sigma.\tau}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : \sigma.\tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : [x \mapsto e_2]\tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \circ e_2 : \texttt{real}}, \circ \in \{+, -, *, /\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \circ e_2 : \texttt{bool}}, \circ \in \{=, <, >, \geq, \leq\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \Pi(n : \texttt{nat}).\texttt{nat} \\ \Gamma \vdash h_i : \Pi(n : \texttt{nat}).(\Pi(l : \texttt{list}(n)).\texttt{nat}) \\ \Gamma \vdash \hat{h_i} : \Pi(n : \texttt{nat}).(\Pi(l : \texttt{list}(n)).\texttt{list}(hnl))\end{array}}{\Gamma \vdash \texttt{rec}(a, (h_1, \ldots, h_n), (\hat{h_1} \ldots \hat{h_n})) : \Pi(n : \texttt{nat}).(\Pi(l : \texttt{list}(n)).\texttt{nat})}$$

FIGURE 2. Typing for dependent type language.

We may express this function in our expression language as follows:

$$\texttt{T} = \texttt{rec}(a, (h_1, h_2), (\hat{h_1}, \hat{h_2}))$$

Then, we want to prove the following claim:

$e = e$

$\mathtt{0} + \mathtt{0} = \mathtt{0},\ \mathtt{0} + \mathtt{1} = 1, \ldots,\ \mathtt{3} + \mathtt{5} = \mathtt{8}, \ldots$
equivalent rules for $-$, $*$, and / operations.

$(n = n) = \mathtt{true}$
$(n = m) = \mathtt{false}$, provided $n,\ m$ distinct numerals.

$\mathtt{if\ \ true\ \ then}\ e_1\ \mathtt{else}\ e_2 = e_1$
$\mathtt{if\ \ false\ \ then}\ e_1\ \mathtt{else}\ e_2 = e_2$

$\lambda x.e = \lambda y.[x \mapsto y]e$, provided y not free in e.

$\lambda x.e_1\ e_2 = [x \mapsto e_2]e_1$
$\mathtt{rec}(\lambda x.a, (\lambda x.\lambda l.h_1, \ldots \lambda x.\lambda l.h_n)(\lambda x.\lambda l.\hat{h_1}, \ldots \lambda x.\lambda l.\hat{h_n})) =$
$\lambda x.(e_1 + \sum_{i=1}^{n}(\mathtt{rec}(\lambda x.a, \sum_{i=1}^{n} \lambda x.\lambda l.h_i, \sum_{i=1}^{n} \lambda x.\lambda l.\hat{h_i})h_i\ \hat{h_i}))$

FIGURE 3. Equational Semantics for the language.

$$
\begin{aligned}
[\![\mathtt{real}]\!] &= \mathbb{R} \\
[\![\mathtt{nat}]\!] &= \mathbb{N} \\
[\![\mathtt{bool}]\!] &= \{true, false\} \\
[\![\Pi x : \tau.\sigma]\!]\eta &= \{f : [\![\tau]\!]\eta \to \bigcup_{a \in [\![\tau]\!]\eta} [\![\sigma]\!]\eta\{x \mapsto a\} \mid \\
&\qquad \forall a \in [\![\tau]\!]\eta,\ f(a) \in [\![\sigma]\!]\eta\{x \mapsto a\}\} \\
[\![\mathtt{list}(e)]\!]\eta &= \Omega_{[\![e]\!]\eta} \\
\forall n \in \mathbb{N},\ &\Omega_n \text{ is the sample space of all lists of length } n.
\end{aligned}
$$

FIGURE 4. Denotational semantics for types.

THEOREM 11. *For all $n \in \mathbb{N},\ l \in \Omega_n$, suppose $[\![h_1]\!]\eta(n)(l)$ and $[\![h_2]\!]\eta(n)(l)$ are equal to the number of elements in l less than and greater than or equal to the head of the l, respectively (that is, they accurately reflect the sizes of the subproblems of quicksort). Likewise, suppose $[\![\hat{h_1}]\!]\eta(n)(l)$ and $[\![\hat{h_2}]\!]\eta(n)(l)$ are equal to lists*

$$
\begin{aligned}
[\![\texttt{0}]\!]\eta = 0,\ [\![\texttt{1}]\!]\eta = 1,\ \ldots \\
[\![x : \tau]\!]\eta &= \eta(x) \\
[\![\lambda(x:\tau).e]\!]\eta &= f : [\![\tau]\!]\eta \to \bigcup_{a \in [\![\tau]\!]\eta} [\![\sigma]\!]\eta\{x \mapsto a\} \\
\text{s.t. } \forall a \in [\![\tau]\!]\eta, f(a) &= [\![e]\!]\eta\{x \mapsto a\} \in [\![\sigma]\!]\eta\{x \mapsto a\} \\
[\![e_1\ e_2]\!]\eta &= [\![e_1]\!]\eta([\![e_2]\!]\eta) \\
[\![e_1 + e_2]\!]\eta &= [\![e_1]\!]\eta + [\![e_2]\!]\eta \\
\text{Similar rules for } -,\ *,\ / \\
[\![\texttt{true}]\!]\eta = true,\ [\![\texttt{false}]\!]\eta &= false \\
[\![e_1 = e_2]\!]\eta &= \begin{cases} true \text{ if } ([\![e_1]\!]\eta = [\![e_2]\!]\eta) \\ false \text{ if } ([\![e_1]\!]\eta \neq [\![e_2]\!]\eta) \end{cases} \\
\text{Similar rules for } <,\ \leq, >, \geq \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\eta &= \begin{cases} [\![e_2]\!]\eta \text{ if } [\![e_1]\!]\eta = true \\ [\![e_3]\!]\eta \text{ if } [\![e_1]\!]\eta = false \end{cases} \\
[\![(e_1, e_2)]\!]\eta &= ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![\texttt{rec}(a, (h_1, \ldots, h_n), (\hat{h_1}, \ldots, \hat{h_n}))]\!]\eta &= rec : \mathbb{N} \to (\bigcup_{n \in \mathbb{N}} \Omega_n \to \mathbb{N}) \text{ s.t.} \\
\forall n \in \mathbb{N},\ \forall l \in \Omega_n, rec(n)(l) &= \sum_{j=0}^{\infty} (\sum_{(n',l') \in S^j(n,l)} [\![a]\!]\eta(n')) \\
S^j \text{ is defined inductively as follows:} \\
S^0(n,l) &= \{(n,l)\} \\
S^{j+1}(n,l) &= \bigcup_{0<i\leq n} \{([\![h_i]\!]\eta(n')(l'), [\![\hat{h_i}]\!]\eta(n')(l')) \mid \\
&\qquad (n',l') \in S^j\}
\end{aligned}
$$

FIGURE 5. Denotational semantics for expressions.

containing the elements of l less than and greater than or equal to the head of l, respectively. Then $[\![T]\!]\eta(n)(l)$ is equal to the cost of performing quicksort on l.

PROOF. By induction on n.

Base case: $n = 1$. For all $l \in \Omega_n$, l is a list of length 1. Then

$$\begin{aligned}
[\![\mathtt{T}]\!]\eta(n)(l) &= \sum_{j=0}^{\infty}\Big(\sum_{(n',l')\in S^j(n,l)} [\![a]\!]\eta(n')\Big) \\
&= [\![a]\!]\eta(n) + \sum_{j=1}^{\infty}\Big(\sum_{(n',l')\in S^j(n,l)} [\![a]\!]\eta(n')\Big) \\
&= [\![a]\!]\eta(n) + 0 = n - 1 = 0,
\end{aligned}$$

since the subproblems of quicksorting a list of length 1 both have size 0. See that this is the actual cost of performing quicksort on a list of length 1, so the claim holds.

Inductive case: Suppose the claim holds for all $k < n$, and let $l \in \Omega_n$. Then

$$\begin{aligned}
[\![\mathtt{T}]\!]\eta(n)(l) &= \sum_{j=0}^{\infty}\Big(\sum_{(n',l')\in S^j(n,l)} [\![a]\!]\eta(n')\Big) \\
&= [\![a]\!]\eta(n) + \sum_{j=1}^{\infty}\Big(\sum_{(n',l')\in S^j(n,l)} [\![a]\!]\eta(n')\Big) \\
&= (n-1) + \sum_{j=0}^{\infty}\Big(\sum_{(n',l')\in S^j([\![h_1]\!]\eta(n),[\![\hat{h_1}]\!]\eta(l))} [\![a]\!]\eta(n')\Big) \\
&\qquad + \sum_{j=0}^{\infty}\Big(\sum_{(n',l')\in S^j([\![h_2]\!]\eta(n),[\![\hat{h_2}]\!]\eta(l))} [\![a]\!]\eta(n')\Big) \\
&= (n-1) + [\![\mathtt{T}]\!]\eta([\![h_1]\!]\eta(n)(l))([\![\hat{h_1}]\!]\eta(n)(l)) \\
&\qquad + [\![\mathtt{T}]\!]\eta([\![h_2]\!]\eta(n)(l))([\![\hat{h_2}]\!]\eta(n)(l))
\end{aligned}$$

By inductive hypothesis,

$$[\![\mathtt{T}]\!]\eta([\![h_1]\!]\eta(n)(l))([\![\hat{h_1}]\!]\eta(n)(l)) \text{ and}$$

$$[\![\mathtt{T}]\!]\eta([\![h_2]\!]\eta(n)(l))([\![\hat{h_2}]\!]\eta(n)(l))$$

are equal to the cost of performing quicksort on $[\![\hat{h_1}]\!]\eta(n)(l)$ and $[\![\hat{h_2}]\!]\eta(n)(l)$—i.e., the left and right sublists of l. Then $[\![\mathtt{T}]\!]\eta(n)(l)$ is equal to the sum of these costs, plus $n-1$; this is the cost of performing quicksort on l.

Therefore, the claim holds for all n. □

Thus, we have a clearer indication that the syntax and denotational semantics defined in this chapter correctly represent Karp's probabilistic recurrence relations.

CHAPTER 6

Conclusion

We have defined three languages which allow us to write and type-check the several types of recurrences described by Karp. The first allows us to describe deterministic recurrences of the form $T(x) = a(x) + T(m(x))$, where a and m are real-valued functions. The second allows us to describe recurrences of this form, but with an added initial condition $T(m) = c$. And the third allows us to describe probabilistic recurrences of the form $T(x) = a(x) + T(h(x)(l))(\hat{h}(x)(l))$, where a is a real-valued function, $h(x)$ is a random variable of type $\Omega_x \to \mathbb{R}$, and $\hat{h}(x)$ is a random variable of type $\prod_{l \in \Omega_x} \Omega_{h(x)(l)}$. Moreover, we have discussed a means of translating between the first and second syntaxes.

1. Future Work

In future work, we hope to tie in the work done in this thesis to the current cost extraction framework developed by Danner et al. [2015]. Its current structure is as follows: a source language program e is mapped by an extraction function to a syntactic cost recurrence E. This syntactic recurrence is then mapped by a denotational semantics to a semantic recurrence $[\![E]\!]$ which describes an upper bound on the cost of e.

$$e \xrightarrow{\text{extr. function}} E \xrightarrow{\text{den. semantics}} [\![E]\!]$$

However, as discussed in the introduction, this denotational semantics does not always offer useful bounds on probabilistic recurrences. We aim to expand the

cost recurrence syntax to support the syntax for probabilistic recurrences offered in this thesis, as well as to include the denotational semantics for that syntax. This will mean that, given a syntactic recurrence E, we can interpret it using either denotational semantics; which semantics we use will be determined by what kind of analysis we wish to make. This will allow us to obtain better bounds on the costs of a wider range of algorithms.

Bibliography

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. page 140?151, 2015. doi: 10.1145/2784731.2784749.

Richard M. Karp. Probabilistic recurrence relations. *Journal of the ACM*, 41(6): 1136–1150, 1994. doi: 10.1145/195613.195632.

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

Joseph Tassarotti and Robert Harper. Verified tail bounds for randomized programs. Preprint, 2017.