A Programming Language for Probabilistic Computation

Sungwoo Park

August 2005
CMU-CS-05-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, co-chair
Sebastian Thrun, co-chair, Stanford University
Geoffrey Gordon
Robert Harper
Norman Ramsey, Harvard University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored by the US Navy under contract no. N6600101C6018, the US Air Force under contract nos. F1962895C0050 and F306029820137, the US Army under contract no. DABT6300C1016, and through a generous grant from the National Science Foundation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Probabilistic language, Probability distribution, Sampling function, Robotics, Computational effect, Monad, Modal logic.

Abstract

As probabilistic computations play an increasing role in solving various problems, researchers have designed *probabilistic languages* to facilitate their modeling. Most of the existing probabilistic languages, however, focus only on discrete distributions, and there has been little effort to develop probabilistic languages whose expressive power is beyond discrete distributions. This dissertation presents a probabilistic language, called *PTP* (*P*robabilis*T*ic *P*rogramming), which supports all kinds of probability distributions.

The key idea behind PTP is to use *sampling functions*, *i.e.*, mappings from the unit interval $(0.0, 1.0]$ to probability domains, to specify probability distributions. By using sampling functions as its mathematical basis, PTP provides a unified representation scheme for probability distributions, without drawing a syntactic or semantic distinction between different kinds of probability distributions.

Independently of PTP, we develop a linguistic framework, called $\lambda_{\bigcirc}$, to account for computational effects in general. $\lambda_{\bigcirc}$ extends a monadic language by applying the possible world interpretation of modal logic. A characteristic feature of $\lambda_{\bigcirc}$ is the distinction between stateful computational effects, called *world effects*, and contextual computational effects, called *control effects*. PTP arises as an instance of $\lambda_{\bigcirc}$ with a language construct for probabilistic choices.

We use a sound and complete translator of PTP to embed it in Objective CAML. The use of PTP is demonstrated with three applications in robotics: robot localization, people tracking, and robotic mapping. Thus PTP serves as another example of high-level language applied to a problem domain where imperative languages have been traditionally dominant.

Acknowledgments

I am grateful to my advisor Frank Pfenning for all the support he gave me during my graduate years. From his unusual patience, hearty encouragement, and eternal cheerfulness, I witnessed the excellent leadership of an academic advisor; from all the technical discussions we had together, I learned everything I know about programming language theory. I am also grateful to my co-advisor Sebastian Thrun for suggesting to me the thesis topic and teaching me robotics.

I thank my thesis committee for their time and involvement. Chapter 2 grew out of numerous discussions with Bob Harper, which were always fun and are still a source of inspirations for me. I am particularly indebted to Norman Ramsey for showing an interest in my work, carefully reading the draft, and writing insightful comments even twice. I also thank Sharon Burks for her patience and assistance in completing the dissertation for the last few months.

I thank my friends at Carnegie Mellon with whom I have shared the experience of graduate school: Hakan Younes, Tiankai Tu, Jonathan Moody, Jeff Polakow, Joshua Dunfield, and Amit Manjhi. I also thank my high school friends who I never thought would live in Pittsburgh during my graduate years: Jaedong Kim, Sunghong Park, and Wootae Kim. I express my sincere gratitude to Eunseok Cha who was unconditionally supportive of me at all the hard times in Pittsburgh. She was also always with me at all the happy times, which I deeply appreciate.

Lastly the dissertation would not have been written without the wholehearted support of my family for the past six years. Ultimately the dissertation is for all of my family and it is our achievement.

Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the design, implementation, and applications of a probabilistic language called *PTP* (*P*robabilis*T*ic *P*rogramming). PTP uses *sampling functions*, *i.e.*, mappings from the unit interval $(0.0, 1.0]$ to probability domains, to specify probability distributions. By using sampling functions in specifying probability distributions, PTP supports all kinds of probability distributions in a uniform manner. The use of PTP is demonstrated with three applications in robotics: robot localization, people tracking, and robotic mapping.

The contribution of this dissertation is three-fold:

- *Sampling functions for specifying probability distributions*. As most of the existing probabilistic languages focus only on discrete distributions, probabilistic computations involving non-discrete distributions have usually been implemented in conventional languages. Sampling functions open a new way to specify all kinds of probability distributions, and thus serve as a mathematical basis for probabilistic languages whose expressive power is beyond discrete distributions.

- *Linguistic framework for computational effects*. We develop a new linguistic framework, called $\lambda_{\bigcirc}$, to account for computational effects in general. $\lambda_{\bigcirc}$ extends the monadic language of Pfenning and Davies [60] by applying the possible world interpretation of modal logic. It distinguishes between stateful computational effects (called *world effects*) and contextual computational effects (called *control effects*), and provides a different view on how to combine computational effects at the language design level. PTP arises as an instance of $\lambda_{\bigcirc}$ with a language construct for probabilistic choices.

- *Applications of PTP in robotics*. In order to execute PTP programs, we use a sound and complete translator of PTP to embed it in Objective CAML. The use of PTP is then demonstrated with three applications in robotics: robot localization, people tracking, and robotic mapping. Thus PTP serves as another example of high-level language applied to a problem domain where imperative languages have been traditionally dominant.

1.1 Motivation

A probabilistic computation is a computation which makes probabilistic choices or whose result is represented with probability distributions. As an alternative paradigm to deterministic computation, it has been used successfully in diverse fields of computer science such as speech recognition [63, 29], natural language processing [11], and robotics [72]. Its success lies in the fact that probabilistic approaches often overcome the practical limitation of deterministic approaches. A trivial example is the problem of testing whether a multivariate polynomial given by a program without branch statements is identically zero or not. It is difficult to find a practical deterministic solution, but there is a simple probabilistic solution: evaluate the polynomial on a randomly chosen input and check if the result is zero.

As probabilistic computations play an increasing role in solving various problems, researchers have also designed *probabilistic languages* to facilitate their implementation [33, 24, 74, 59, 64, 43, 53]. A probabilistic language treats probability distributions as built-in datatypes and thus abstracts from representation schemes, *i.e.*, data structures for representing probability distributions. For example, a conventional language may be extended with an abstract datatype for probability distributions, which is specified by a certain choice of representation scheme and a set of operations on probability distributions. As a result, it allows programmers to concentrate on how to formulate probabilistic computations at the level of probability distributions rather than representation schemes. When translated in a probabilistic language (by programmers), such a formulation usually produces concise and elegant code.

A typical probabilistic language supports at least discrete distributions, for which there exists a representation scheme sufficient for all practical purposes: a set of pairs consisting of a value from the probability domain and its probability. We can use such a probabilistic language for those problems involving only discrete distributions. If non-discrete distributions are involved, however, we usually use a conventional language for the sake of efficiency, assuming a specific kind of probability distributions (*e.g.*, Gaussian distributions) or choosing a specific representation scheme (*e.g.*, a set of samples from the probability distribution). For this reason, there has been little effort to develop probabilistic languages whose expressive power is beyond discrete distributions.

The unavailability of such probabilistic languages means that when implementing a probabilistic computation involving non-discrete distributions, we have to resort to a conventional language. Thus we wish to develop a probabilistic language supporting all kinds of probability distributions — *discrete distributions, continuous distributions, and even those belonging to neither group.* Furthermore we wish to draw no distinction between different kinds of probability distributions, both syntactically and semantically, so that we can achieve a uniform framework for probabilistic computation. Such a probabilistic language can have a significant practical impact, since once formulated at the level of probability distributions, any probabilistic computation can be directly translated into code.

Below we present an example that illustrates the disadvantage of conventional languages in implementing probabilistic computations and also motivates the development of PTP.

Notation

If a variable x ranges over the domain of a probability distribution P, then $P(x)$ means, depending on the context, either the probability distribution itself (as in "probability distribution $P(x)$") or the probability of a particular value x (as in "probability $P(x)$"). We write $P(x)$ for probability distribution P when we want to emphasize the use of variable x. If we do not need a specific name for a probability distribution, we use $Prob$ (as in "probability distribution $Prob(x)$").

Similarly $P(x|y)$ means either the conditional probability P itself or the probability of x conditioned on y. We write P_y or $P(\cdot|y)$ for the probability distribution conditioned on y.

$U(0.0, 1.0]$ denotes a uniform distribution over the unit interval $(0.0, 1.0]$.

A motivating example for PTP

A *Bayes filter* [28] is a popular solution to a wide range of state estimation problems. It estimates the state s of a system from a sequence of *actions* and *measurements*, where an action a induces a change to the state and a measurement m gives information on the state. At its core, a Bayes filter computes a probability distribution $Bel(s)$ of the state according to the following update equations:

$$Bel(s) \leftarrow \int \mathcal{A}(s|a,s')Bel(s')ds' \tag{1.1}$$

$$Bel(s) \leftarrow \eta \mathcal{P}(m|s)Bel(s) \tag{1.2}$$

$\mathcal{A}(s|a,s')$ is the probability that the system transitions to state s after taking action a in another state s', $\mathcal{P}(m|s)$ the probability of measurement m in state s, and η a normalizing constant ensuring $\int Bel(s)ds = 1.0$. The update equations are formulated at the level of probability distributions in the sense that they do not assume a particular representation scheme.

Unfortunately the update equations are difficult to implement for arbitrary probability distributions. When it comes to implementation, therefore, we usually simplify the update equations by making additional assumptions on the system or choosing a specific representation scheme. For example, with the assumption that Bel is a Gaussian distribution, we obtain a variant of the Bayes filter called a *Kalman filter* [79]. If Bel is approximated with a set of samples, we obtain another variant called a *particle filter* [15].

Even these variants of the Bayes filter are, however, not trivial to implement in conventional languages. For example, a Kalman filter requires various matrix operations including matrix inversion. A particle filter manipulates weights associated with individual samples, which often results in complicated code. Since conventional languages can only simulate probability distributions, it is also difficult to figure out the intended meaning of the code, namely the update equations for the Bayes filter.

An alternative approach is to use an existing probabilistic language after discretizing all probability distributions. This idea is appealing in theory, but impractical for two reasons. First, given a probability distribution, it may not be easy to choose an appropriate subset of its support upon which discretization is performed. For example, in order to discretize a Gaussian distribution (whose support is $(-\infty, \infty)$), we need to choose a threshold for probabilities so that discretization is confined to an interval of finite length; for an arbitrary probability distribution, such a threshold can be computed only by examining its entire probability domain. Even when the subset of its support is fixed in advance, the process of discretization may incur a considerable amount of programming. For example, Fox *et al.* [20] develop two non-trivial techniques (specific to their applications) for the sole purpose of efficiently manipulating discretized probability distributions. Second some probability distributions cannot be discretized in any meaningful way. An example is probability distributions over probability distributions or functions, which do occur in real applications (Chapter 5 presents such an example).

If there were a probabilistic language supporting all kinds of probability distributions, we could implement the update equations with much less effort. PTP is a probabilistic language designed with these goals in mind.

1.2 Previous work

There are a number of probabilistic languages that focus on discrete distributions. Such a language usually provides a probabilistic construct that is equivalent to a binary choice construct. Saheb-Djahromi [69] presents a probabilistic language with a binary choice construct $(p_1 \rightarrow e_1, p_2 \rightarrow e_2)$ where $p_1 + p_2 = 1.0$.[1] Koller, McAllester, and Pfeffer [33] present a first order functional language with a coin toss construct $\mathsf{flip}(p)$ where p is a probability in $(0.0, 1.0)$. Pfeffer [59] generalizes the coin toss construct to a multiple choice construct $\mathsf{dist}\,[p_1 : e_1, \cdots, p_n : e_n]$ where $\sum_i p_i = 1.0$. Gupta, Jagadeesan, and Panangaden [24] present a stochastic concurrent constraint language with a probabilistic choice construct $\mathsf{choose}\ x\ \mathsf{from}\ Dom\ \mathsf{in}\ e$ where Dom is a finite set of real numbers. Ramsey and Pfeffer [64] present a stochastic lambda calculus with

[1] In this section, p (with or without indices) stands for probabilities, e program fragments, and v values.

a binary choice construct $\mathsf{choose}\ p\ e_1\ e_2$. All these constructs, although in different forms, are equivalent to a binary choice construct and have the same expressive power.

An easy way to process a binary choice construct (or an equivalent) during a computation is to generate a sample from the probability distribution it denotes, as in the above probabilistic languages. Another way is to return an accurate representation of the probability distribution itself, by enumerating all elements in its support along with their probabilities. Pless and Luger [61] present an extended lambda calculus which uses a probabilistic construct of the form $\sum_i e_i : p_i$ where $\sum_i p_i = 1.0$. A program denoting a probability distribution computes to a normal form $\sum_i v_i : p_i$, which is an accurate representation of the probability distribution. Jones [30] presents a metalanguage with a binary choice construct $e_1\ \mathsf{or}_p\ e_2$. Its operational semantics uses a judgment $e \Rightarrow \sum p_i v_i$. Mogensen [43] presents a language for specifying die-rolls. Its denotational semantics (called probability semantics) is formulated in a similar style, directly in terms of probability measures.

Jones and Mogensen also provide an equivalent of a recursion construct which enables programmers to specify discrete distributions with infinite support (*e.g.*, geometric distribution). Such a probability distribution is, however, difficult to represent accurately because of an infinite number of elements in its support. For this reason, Jones assumes $\sum p_i \leq 1.0$ in the judgment $e \Rightarrow \sum p_i v_i$ and Mogensen uses partial probability distributions in which the sum of probabilities may be less than 1.0. The intuition is that a finite recursion depth is used so that some elements in the support are omitted in the enumeration.

There are a few probabilistic languages supporting continuous distributions. Kozen [34] investigates the semantics of probabilistic while programs. A random assignment $x := \mathsf{random}$ assigns a random number to variable x. Since it does not assume a specific probability distribution for the random number generator, the language serves only as a framework for probabilistic languages. Thrun [73, 74] extends C++ with probabilistic data types which are created from a template prob<*type*>. Although the language, called *CES*, supports common continuous distributions, its semantics is not formally defined. Our work is originally motivated by the desire to develop a probabilistic language that is as expressive as CES and also has a formal semantics.

1.3 Sampling functions as the mathematical basis

The expressive power of a probabilistic language is determined to a large extent by its mathematical basis. That is, the set of probability distributions expressible in a probabilistic language is determined principally by mathematical objects used in specifying probability distributions. Since we intend to support all kinds of probability distributions without drawing a syntactic or semantic distinction, we cannot choose what is applicable only to a specific kind of probability distributions. Examples are probability mass functions which are specific to discrete distributions, probability density functions which are specific to continuous distributions, and cumulative distribution functions which assume an ordering on each probability domain.

Probability measures [65] are a possibility because they are synonymous with probability distributions. A probability measure μ over a domain $\mathcal{D}$ is a mapping satisfying the following conditions:

- $\mu(\emptyset) = 0$.
- $\mu(\mathcal{D}) = 1$.
- For a countable disjoint union $\cup_i D_i$ of subsets D_i of $\mathcal{D}$,

  $$\mu(\cup_i D_i) = \textstyle\sum_i \mu(D_i)$$

  where $\cup_i D_i$ is required to be a subset of $\mathcal{D}$.

Conceptually it maps the set of subsets of $\mathcal{D}$ (or, the set of events on $\mathcal{D}$) to probabilities in $[0.0, 1.0]$. Probability measures are, however, not a practical choice as the mathematical basis because they are difficult to represent if the domain in infinite. As an example, consider a continuous probability distribution P of the position of a robot in a two-dimensional environment. (Since P is continuous, the domain is infinite even if the environment is physically finite.) The probability measure μ corresponding to P should be able to calculate a probability for any given part of the environment (as opposed to a particular spot in the environment) — whether it is a contiguous region or a collection of disjoint regions, or whether it rectangular or oval-shaped. Thus finding a suitable representation for μ involves the problem of representing an arbitrary part of the environment, and is thus far from a routine task.

The main idea of our work is that we can specify a probability distribution by answering *"How can we generate samples from it?"*, or equivalently, by providing *a sampling function* for it. A sampling function is defined as a mapping from the unit interval $(0.0, 1.0]$ to a probability domain $\mathcal{D}$. Given a random number drawn from $U(0.0, 1.0]$, it returns a sample in $\mathcal{D}$, and thus specifies a unique probability distribution. In this way, random numbers serve as the source of probabilistic choices.

In specifying how to generate samples, we wish to exploit sampling techniques developed in simulation theory [10], most of which consume multiple (independent) random numbers to produce a single sample. To this end, we use a generalized notion of sampling function which maps $(0.0, 1.0]^\infty$ to $\mathcal{D} \times (0.0, 1.0]^\infty$ where $(0.0, 1.0]^\infty$ denotes an infinite product of $(0.0, 1.0]$. Operationally a sampling function now takes as input an infinite sequence of random numbers drawn independently from $U(0.0, 1.0]$, consumes zero or more random numbers, and returns a sample with the remaining sequence. This generalization of the notion of sampling function is acceptable arithmetically (but not measure-theoretically). For example, we can use the technique of expanding a single real number in $(0.0, 1.0]$ into an infinite sequence of real numbers in $(0.0, 1.0]$ by taking even and odd bits of a binary representation of a given real number to produce two real numbers and repeating the procedure.

As the mathematical basis of PTP, we choose sampling functions, which overcome the problem with probability measures: they are applicable to all kinds of probability distributions, and are also easy to represent because a global random number generator (which generates as many random numbers as necessary from $U(0.0, 1.0]$) supplants the use of infinite sequences of random numbers. As a comparison with probability measures, consider the probability distribution P of the position of a robot discussed above. In devising a sampling function for P, we only have to construct an algorithm that probabilistically generates possible positions of the robot; hence we do not need to consider the problem of representing an arbitrary part of the environment (which is essential in the case of probability measures). Intuitively it is easier to both formalize and answer *"Where is the robot likely to be?"* than *"How likely is the robot to be in a given region?"*.

The use of sampling functions as the mathematical basis leads to three desirable properties of PTP. First it provides a unified representation scheme for probability distributions: we no longer distinguish between discrete distributions, continuous distributions, and even those belonging to neither group. Such a unified representation scheme is difficult to achieve with other candidates for the mathematical basis. Second it enjoys rich expressiveness: we can specify probability distributions over infinite discrete domains, continuous domains, and even unusual domains such as infinite data structures (*e.g.*, trees) and cyclic domains (*e.g.*, angular values). Third it enjoys high versatility: there can be more than one way to specify a probability distribution, and the more we know about it, the better we can encode it. Section 3.2 demonstrates these properties with various examples written in PTP.

Data abstraction for probability distributions

In PTP, a sampling function is represented by a probabilistic computation that consumes zero or more random numbers (rather than a single random number) drawn from $U(0.0, 1.0]$. In the context of data abstraction, it means that a probability distribution is *constructed* from such a probabilistic computation. The expressive power of PTP allows programmers to construct (or encode) all kinds of probability distributions in a uniform way. Equally important is, however, the question of how to *observe* (or reason about) a given probability distribution, *i.e.*, how to get information out of it, through various queries. Since a probabilistic computation in PTP only describes a procedure for generating samples, the only way to observe a probability distribution is by generating samples from it. As a result, PTP is limited in its support for queries on probability distributions. For example, it does not permit a precise implementation of such queries as means, variances, and probabilities of specific events.

PTP alleviates this limitation by exploiting the Monte Carlo method [40], which approximately answers a query on a probability distribution by generating a large number of samples and then analyzing them. As an example, consider a (continuous) probability distribution P of the pose (*i.e.*, position and orientation) of a robot in a two-dimensional environment. Here are a few queries on P all of which can be answered approximately:

- *Draw a sample of robot pose at random.*
- *What is the expected (average) pose of the robot?*
- *What is the probability that the robot is facing within five degrees of due east?*
- *What is the probability that the robot is in Peter's office?*
- *Under the assumption that the robot is in Peter's office, what is the probability that the robot is within two feet of the door?"*

These queries can be answered approximately by repeatedly performing the probabilistic computation associated with P and then analyzing resultant samples. For example, the last query can be answered as follows:

1. Generate samples from P.
2. Filter out those samples indicating that the robot is not in Peter's office.
3. Count the number of samples indicating that the robot is within two feet of the door, and divide it by the total number of remaining samples.

Certain queries on probability distributions are, however, difficult to answer even approximately by the Monte Carlo method. For example, the following queries are difficult to answer approximately by a simple analysis of samples:

- *What is the most likely position of the robot?*
- *In what room is the robot most likely to be when the number of rooms is unknown?*

Due to the nature of the Monte Carlo method, the cost of answering a query is proportional to the number of samples used in the analysis. The cost of generating a single sample is determined by the specific procedure chosen by programmers, rather than by the probability distribution itself from which to draw samples. For example, a geometric distribution can be encoded with a recursive procedure which simulates coin tosses until a certain outcome is observed, or by a simple transformation (called the *inverse transform method*) which requires only a single random number. These two methods of encoding the same probability distribution differ in the cost of generating a single sample and hence in the cost of answering the same query by the Monte Carlo method. For a similar reason, the accuracy of the result of the Monte Carlo method, which improves with the number of samples, is also affected by the procedure chosen by programmers.

Measure-theoretic view of sampling functions

The accepted mathematical basis of probability theory is measure theory [65], which associates every probability distribution with a unique probability measure. We give a summary of measure theory before discussing the connection between sampling functions and measure theory. In the discussion below, sampling functions refer to those taking $(0.0, 1.0]$ as input, rather than generalized ones taking $(0.0, 1.0]^\infty$ as input.

- *Measurable sets* of a space $\mathcal{D}$ are subsets of $\mathcal{D}$.
- A *measurable space* $\mathsf{M}(\mathcal{D})$ is a collection of measurable sets of $\mathcal{D}$ such that:
  - $\mathcal{D} \in \mathsf{M}(\mathcal{D})$.
  - If $S \in \mathsf{M}(\mathcal{D})$, then $\mathcal{D} - S \in \mathsf{M}(\mathcal{D})$. That is, $\mathsf{M}(\mathcal{D})$ is closed under complement.
  - For a countable collection of measurable sets $S_i \in \mathsf{M}(\mathcal{D})$, it holds $\cup_i S_i \in \mathsf{M}(\mathcal{D})$. That is, $\mathsf{M}(\mathcal{D})$ is closed under countable union.
- A *measurable function* f from $\mathcal{D}$ to $\mathcal{E}$ is a mapping from $\mathsf{M}(\mathcal{D})$ to $\mathsf{M}(\mathcal{E})$ such that if $S \in \mathsf{M}(\mathcal{E})$, then $f^{-1}(S) \in \mathsf{M}(\mathcal{D})$.
- A *measure* μ over $\mathsf{M}(\mathcal{D})$ is a mapping from $\mathsf{M}(\mathcal{D})$ to $[0.0, \infty]$ such that:
  - $\mu(\emptyset) = 0$.
  - For a countable disjoint union $\cup_i S_i$ of measurable sets $S_i \in \mathsf{M}(\mathcal{D})$, it holds $\mu(\cup_i S_i) = \Sigma_i \mu(S_i)$.
- A *probability measure* μ over $\mathsf{M}(\mathcal{D})$ satisfies $\mu(\mathcal{D}) = 1$.
- A *Lebesgue measure* ν over the unit interval $(0.0, 1.0]$ is a probability measure such that $\nu(S)$ is equal to the total length of intervals in S.

Measure theory allows certain (but not all) sampling functions to specify probability distributions. Consider a sampling function f from $(0.0, 1.0]$ to $\mathcal{D}$. While it is introduced primarily as a mathematical function, f may be interpreted as a measurable function as well, in which case it defines a unique probability measure μ over $\mathsf{M}(\mathcal{D})$ such that

$$\mu(S) = \nu(f^{-1}(S))$$

where ν is a Lebesgue measure over the unit interval. The intuition is that S, as an event, is assigned a probability equal to the size of it inverse image under f.

This dissertation does not investigate measure-theoretic properties of sampling functions definable in PTP. If a probabilistic computation expressed in PTP consumes at most one random number (drawn from $U(0.0, 1.0]$), it is easy to identify a corresponding sampling function. If more than one sample is consumed, however, it is not always obvious how to construct such a sampling function. In fact, the presence of fixed point constructs in PTP (for recursive computations which can consume an arbitrary number of random numbers) makes it difficult even to define measurable spaces to which sampling functions map the unit interval, since fixed point constructs use domain-theoretic structures, rather than measure-theoretic structures, in order to solve resultant recursive equations.

Every probabilistic computation expressed in PTP is easily translated into a generalized sampling function (which takes $(0.0, 1.0]^\infty$ as input). It is, however, unknown if generalized sampling functions definable in PTP are all measurable. Also unknown is if generalized sampling functions are measure-theoretically equivalent to ordinary sampling functions (*i.e.*, if a measurable function from $(0.0, 1.0]^\infty$ to $\mathcal{D} \times (0.0, 1.0]^\infty$ determines a unique measurable function from $(0.0, 1.0]$ to $\mathcal{D}$). Nevertheless generalized sampling functions definable in PTP are shown to be closely connected with sampling techniques from simulation theory, which, like measure theory, are widely agreed to be a form of probabilistic computation and PTP is designed to support. A further discussion is found in Section 3.3.

1.4 Linguistic framework for PTP

We develop PTP as a functional language extending the λ-calculus, rather than an imperative language or a library embedded in an existing conventional language. We decide to use a monadic syntax for probabilistic computations. The decision is based upon two observations. First sampling functions are operationally equivalent to probabilistic computations in that they describe procedures for generating samples from infinite sequences of random numbers. Second sampling functions form a *state monad* [44, 45, 64] whose set of states is $(0.0, 1.0]^\infty$. These two observations imply that if we use a monadic syntax for probabilistic computations, it becomes straightforward to interpret probabilistic computations in terms of sampling functions. The monadic syntax treats probability distributions as first-class values and offers a clean separation between regular values and probabilistic computations.

Instead of designing a monadic syntax specialized for sampling functions, we begin by developing a linguistic framework $\lambda_\bigcirc$ which accounts for computational effects in general. $\lambda_\bigcirc$ does not borrow its syntax from Moggi's monadic metalanguage λ_{ml} [44, 45]. Instead it extends the monadic language of Pfenning and Davies [60], which is a reformulation of λ_{ml} from a modal logic perspective. $\lambda_\bigcirc$ may be thought of as their monadic language combined with the possible world interpretation [35] of modal logic.

A characteristic feature of $\lambda_\bigcirc$ is that it classifies computational effects into two kinds: world effects and control effects. World effects are stateful computational effects such as mutable references and input/output; control effects are contextual computational effects such as exceptions and continuations. Probabilistic choices are a particular case of world effect, and PTP arises as an instance of $\lambda_\bigcirc$ with a language construct for consuming (or drawing) random numbers from $U(0.0, 1.0]$.

1.5 Applications to robotics

Instead of implementing PTP as a complete programming language of its own, we embed it in an existing functional language by building a translator. Specifically we extend the syntax of Objective CAML [2] to incorporate the syntax of PTP, and then translate language constructs of PTP back into the original syntax. The translator is sound and complete in the sense that both type and reducibility of any program in PTP, whether well-typed/reducible or ill-typed/irreducible, are preserved when translated in Objective CAML.

An important part of our work is to demonstrate the use of PTP by applying it to real problems. As the main testbed, we choose *robotics* [72]. It offers a variety of real problems that necessitate probabilistic computations over continuous distributions. We use PTP for three applications in robotics: robot localization [72], people tracking [50], and robotic mapping [75]. In each case, the state of a robot is represented with a probability distribution, whose update equation is formulated at the level of probability distributions and translated directly in PTP. All experiments in our work have been carried out with real robots.

A comparison between our robot localizer and another written in C gives evidence that the benefit of implementing probabilistic computations in PTP, such as readability and conciseness of code, can outweigh its disadvantage in speed (see Section 5.5 for details). Thus PTP serves as another example of high-level language whose power is well exploited in a problem domain where imperative languages have been traditionally dominant.

1.6 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents the linguistic framework $\lambda_{\bigcirc}$ to be used for PTP. Chapter 3 presents the syntax, type system, and operational semantics of PTP. Chapter 4 describes the translator of PTP in Objective CAML. Chapter 5 presents three applications of PTP in robotics. Chapter 6 concludes.

Chapter 2

Linguistic Framework

This chapter presents our linguistic framework $\lambda_{\bigcirc}$ to be used for PTP. $\lambda_{\bigcirc}$ is an extension of the λ-calculus (with a modality $\bigcirc$) which accounts for computational effects in general. In developing $\lambda_{\bigcirc}$, we are interested in modeling such computational effects as input/output, mutable references, and continuations. We view probabilistic choices as a particular case of computational effect, and PTP arises as an instance of $\lambda_{\bigcirc}$ with a language construct for probabilistic choices.

Key concepts used in the development of $\lambda_{\bigcirc}$ are as follows:

- *Segregation of world effects and control effects.* $\lambda_{\bigcirc}$ classifies computational effects into two kinds: stateful world effects and contextual control effects. The distinction makes it easy to combine computational effects at the language design level.

- *Possible world interpretation of modal logic.* $\lambda_{\bigcirc}$ uses modal logic [12] to characterize world effects, and relates modal logic to world effects by the possible world interpretation [35]. As a result, the notion of world in "world effects" coincides with the notion of world in the "possible world interpretation." In formulating the logic for $\lambda_{\bigcirc}$, we use the judgmental style of Pfenning and Davies [60].

At its core, $\lambda_{\bigcirc}$ applies the possible world interpretation to the monadic language of Pfenning and Davies [60], which uses *lax logic* [19, 7] in the judgmental style to reformulate Moggi's monadic metalanguage λ_{ml} [44, 45]. The monadic language of Pfenning and Davies analyzes computational effects only at an abstract level from a proof-theoretic perspective, and does not readily extend to a programming language with computational effects. $\lambda_{\bigcirc}$ is an attempt to extend their monadic language with an operational semantics so as to support concrete notions of computational effect. The key idea is to combine the possible world interpretation and the judgmental style in such a way that the accessibility relation (which is an integral part of the possible world interpretation) is not used in inference rules (unlike the system of modal logic of Simpson [71], for example).

Although $\lambda_{\bigcirc}$ is not specific to probabilistic computations and the development of $\lambda_{\bigcirc}$ is thus optional for the purpose of designing PTP, we investigate $\lambda_{\bigcirc}$ to better explain the logical foundation of PTP. As the definition of PTP in Chapter 3 is self-contained, this chapter can be skipped without loss of continuity by those readers who want to understand only PTP.

2.1 Computational effects in $\lambda_{\bigcirc}$

This section gives a definition of computational effects. The clarification of the notion of computational effect may appear to be of little significance (because we already know what is called computational effects and how they work), but it has a profound impact on the overall design of $\lambda_{\bigcirc}$. This section also gives an overview of $\lambda_{\bigcirc}$ at an abstract level (*i.e.*, without its syntax and semantics).

Definition of computational effects

In the context of functional languages, computation effects are usually defined as what destroys the "purity" of functional languages. Informally the purity of a functional language means that every function in it denotes a mathematical function, *i.e.*, a black box converting a valid argument into a unique outcome. For example, a function `fn x => x + !y` in ML does not denote a mathematical function because its outcome depends on the content of reference `y` as well as argument `x`; hence we conclude that mutable references are computational effects. Other examples of computational effects include input/output, exceptions, continuations, non-determinism, concurrency, and probabilistic choices.

The notion of purity, however, is subtle and there is no universally accepted definition of purity. Sabry [67] shows that common criteria for purity, such as soundness of the β-equational axiom, confluence (the Church-Rosser property or independence of order of evaluation), and preservation of observational equivalences, are incomplete in that either they fail to hold in some pure functional languages or they continue to hold in some impure functional languages (referential transparency is not considered because it does not have a universally accepted definition). He proposes a definition of purity based upon independence of reduction strategies, but this definition has a drawback that a given functional language must have implementations of three reduction strategies, namely, call-by-value, call-by-need, and call-by-name.

As a result, the definition of computational effects as what destroys the purity of functional languages is ambiguous, and some concepts are called computational effects without any justification. For example, non-termination is called a computational effect only by convention (as a special kind of computational effect which is not observable). At the same time, one may argue that non-termination is not a computational effect because the use of pointed types (*i.e.*, types augmented with a bottom element $\perp$ denoting non-termination) preserves the property of mathematical functions.

A definition of computational effects is not necessary in designing a functional language, such as ML and Scheme, that allows any program fragment to produce computational effects. It is, however, crucial to the design of a functional language, such as Haskell 98[1] [55] (and $\lambda_{\bigcirc}$), that subsumes a sublanguage for computational effects, since a criterion for computational effects determines features supported by the sublanguage. The case of Haskell illustrates the importance of a proper definition of computational effects, and also inspires our definition of computational effects.

Computational effects in Haskell

Since their introduction to the programming language community, monads [44, 45] have been considered as an elegant means of structuring functional programs and incorporating computational effects into functional languages [76, 77]. A good example of a functional language that makes extensive use of monads in its design is Haskell. At the programming level, it provides a type class `Monad` to facilitate modular programming; at the language design level, it provides a built-in `IO` monad for producing computational effects without compromising its properties as a pure functional language.

Haskell does not assume a particular definition of computational effects. Instead it implicitly identifies computational effects with monads and confines all kinds of computational effects to the `IO` monad [56, 58] (or a similar one such as the `ST` monad). Thus Haskell conceptually consists of two sublanguages: a functional sublanguage which never produces computational effects, and a monadic sublanguage which is formed by the `IO` monad.

[1]Abbreviated as Haskell henceforth.

The identification between computational effects and monads may appear to be innocuous, perhaps because of the success of monads as a means of modeling different computational effects in a uniform manner. When all kinds of computational effects are present together, however, the identification becomes problematic because monads do not combine well with each other [32, 31, 39]. Haskell uses the `IO` monad for all kinds of computational effects without explicitly addressing this difficulty.

The identification also enforces unconventional treatments of some computational effects. For example, it disallows exceptions for the functional sublanguage, which would be useful for handling division by zero or pattern-match failures. It also disallows continuations for the functional sublanguage, which would be useful for implementing advanced control constructs such as non-local exits and co-routines. Hence the identification significantly limits the practical utility of exceptions and continuations. For this reason, an extension of Haskell proposed by Peyton Jones *et al.* [57] allows exceptions not for the monadic sublanguage but for the functional sublanguage, thereby deviating from the identification between computational effects and monads.

Our view is that computational effects are not identified with monads and that the identification between computational effects and monads in Haskell is a consequence of lack of a proper definition of computational effects. The capability of monads to model all kinds of computational effects may be the rationale for the identification, but it does not really warrant the identification; rather it only implies that monads are a particular tool for studying the denotational semantics of computational effects.

As an example, consider the set monad for modeling non-determinism [76].[2] The set monad is suitable for specifying the denotational semantics of a non-deterministic language (which has a non-deterministic choice construct), since a program can be translated into a set enumerating all possible outcomes. The set monad does not, however, lend itself to the operational design of a non-deterministic language, in which a program returns a single outcome, instead of the set of all possible outcomes, after producing computational effects. Therefore the set monad is useful for developing the denotational semantics (and also possibly the syntax) of a non-deterministic language, but not for implementing it operationally. In fact, if the set monad was enough for implementing a non-deterministic language operationally, we could argue that the built-in `IO` monad is unnecessary in Haskell because we can instantiate the type class `Monad` to mimic all computational effects supported by the `IO` monad. Thus the main lesson learned from Haskell is that modeling a computational effect is a separate issue from implementing it operationally.

Another lesson learned from Haskell is that as its implementation is based upon a state monad, the `IO` monad is suitable for *stateful* computational effects such as mutable references and input/output, but not compatible with *contextual* computational effects such as exceptions and continuations. That is, while stateful computational effects may well be identified with the `IO` monad, contextual computational effects do not need to be restricted to the monadic sublanguage. Our definition of computational effects captures the distinction between these two kinds of computational effects, calling the former *world effects* and the latter *control effects*.

World effects and control effects

We directly define computational effects without relying on another notion such as purity of functional languages. A central assumption is that the run-time system consists of a program and a world. A program is subject to a set of reduction rules. For example, a program in the λ-calculus runs by applying the β-reduction rule. A world is an object whose behavior is specified by the programming environment rather than by reduction rules. For example, a keyboard buffer can be part of a world such that a keystroke or a read operation changes its contents. In contrast, a heap is not part of a world because it is just a convenience for implementing reduction rules. That is, we can implement all reduction rules without using heaps at all.

[2] If the reader holds the view that computational effects and monads are identified, this example may well be hard to follow!

When an external agent or a program interacts with a world and causes a transition to another world, we say that a world effect occurs. For example, if a keyboard buffer is part of a world, a keystroke by a user or a read operation by a program changes its contents and thus causes a world effect. As another example, if a store for mutable references is part of a world, an operation to allocate, dereference, or deallocate references interacts with the world and thus causes a world effect.

When a program undergoes a change that no sequence of reduction rules can induce, we say that a control effect occurs. For example, if the β-reduction rule is the only reduction rule, raising an exception causes a control effect because in general, it induces a change that is independent of the β-reduction rule. For a similar reason, capturing and throwing continuations cause control effects. Note that the concept of control effect is relative to the set of "basic" reduction rules assumed by the run-time system. One could imagine a run-time system with built-in reduction rules for exceptions, in which case raising an exception would not be regarded as a control effect.

Thus world effects and control effects have fundamentally different characteristics and are realized in different ways. World effects are realized by specifying a world structure — empty world structure if there are no world effects, keyboard buffer and display window for input/output, store for mutable references, and so on. Control effects are realized by introducing program transformation rules (that cannot be defined in terms of existing reduction rules). Since world structures and program transformation rules are concerned with different parts of the run-time system, world effects and control effects are treated in orthogonal ways.

The distinction between world effects and control effects makes it easy to combine computational effects at the language design level. Different world effects are combined by merging corresponding world structures. For example, a world structure with a keyboard buffer and display window and a store realizes both input/output and mutable references. There is no need to explicitly combine control effects with other computational effects, since control effects become pervasive once corresponding program transformation rules are introduced.

World effects are further divided into *internal* world effects and and *external* world effects. An internal world effect is always caused by a program and is ephemeral in the sense that the change it makes to a world can be undone by the run-time system. An example is to allocate new references, which can be later reclaimed by the run-time system. An external world effect is caused either by an external agent, affecting a program, or by a program, affecting an external agent. It is perpetual in the sense that the change it makes to a world cannot be undone by the run-time system. An example is to use keyboard input or to send output to a printer — once you type a password to a malicious program or print it on a public printer, there is no going back from the catastrophic consequence!

While internal world effects occur within the run-time system, external world effects involve interactions with external agents. In this regard, all external world effects are examples of concurrency in the presence of external agents. $\lambda_{\bigcirc}$ is not intended to model external agents, and we restrict ourselves to internal world effects in developing $\lambda_{\bigcirc}$.

From Haskell to $\lambda_{\bigcirc}$

As mentioned earlier, Haskell conceptually consists of two sublanguages: 1) a functional sublanguage which is essentially the λ-calculus and never produces computational effects; 2) a monadic sublanguage which is formed by the `IO` monad and produces both world effects and control effects. Peyton Jones [58] clarifies the distinction between the two sublanguages with a two-level semantics: an inner denotational semantics for the functional sublanguage and an outer transition (operational) semantics for the monadic sublanguage.

As control effects do not need to be restricted to the monadic sublanguage, we consider a variant of Haskell that allows both its functional and monadic sublanguages to produce control effects. In comparison with Haskell, this variant has a disadvantage that a function may not denote a mathematical function, but it overcomes the limitation of Haskell in dealing with control effects.

$\lambda_{\bigcirc}$ can be thought of as a reformulation of the variant of Haskell from a logical perspective. It has two syntactic categories: *terms* and *expressions*. Terms form a sublanguage which subsumes the λ-calculus and is allowed to produce only control effects; expressions forms another sublanguage which is allowed to produce both world effects and control effects. The logic behind the definition of expressions is the same as the logic underlying monads, namely lax logic [7]. Thus, like the monadic sublanguage of Haskell, expressions in $\lambda_{\bigcirc}$ enforce the monadic syntax (with the modality $\bigcirc$).

2.2 Logical preliminaries

$\lambda_{\bigcirc}$ has a firm logical foundation, providing a logical analysis of computational effects. This section explains those concepts from logic that play key roles in the development of $\lambda_{\bigcirc}$.

2.2.1 Curry-Howard isomorphism and judgmental formulation

The Curry-Howard isomorphism [27] is a principle connecting logic and programming languages. It states that propositions in logic correspond to types in programming languages (*propositions-as-types* correspondence) and that proofs in logic correspond to programs in programming languages (*proofs-as-programs* correspondence). Given a formulation of logic, it systematically derives the type system and reduction rules of a corresponding programming language. The development of $\lambda_{\bigcirc}$ follows the same pattern: we first formulate the logic for $\lambda_{\bigcirc}$, and then apply the Curry-Howard isomorphism to obtain the type system and reduction rules.

The logic for $\lambda_{\bigcirc}$ is formulated in the judgmental style of Pfenning and Davies [60]. A judgmental formulation of logic adopts Martin-Löf's methodology of distinguishing between *propositions* and *judgments* [42]. It differs from a traditional formulation which relies solely on propositions. Below we review results from Pfenning and Davies [60].

Propositions and judgments

In a judgmental formulation of logic, a proposition is an object of verification whose truth is checked by *inference rules*, whereas a judgment is an object of knowledge which becomes evident by a *proof*. Examples of propositions are '1 + 1 is equal to 0' and '1 + 1 is equal to 2', both under inference rules based upon arithmetic. Examples of judgments are "'1 + 1 is equal to 0' is true", for which there is no proof, and "'1 + 1 is equal to 2' is true", for which there is a proof.

To clarify the difference between propositions and judgments, consider a statement 'the moon is made of cheese.' The statement is not yet an object of verification, or a proposition, since there is no way to check its truth. It becomes a proposition when an inference rule is given, for example, (written in a pedantic way) "'the moon is made of cheese' is true if 'the moon is greenish white and has holes in it' is true." Now we can attempt to verify the proposition, for example, by taking a picture of the moon. That is, we still do not know whether the proposition is true or not, but by virtue of the inference rule, we know at least what counts as a verification of it. If the picture indeed shows that the moon is greenish white and has holes in it, the inference rule makes evident the judgment "'the moon is made of cheese' is true." Now we know "'the moon is made of cheese' is true" by the proof consisting of the picture and the inference rule. Thus a proposition is an object of verification which may or may not be true, whereas a judgment is an object of knowledge which we either know or do not know.

As a more concrete example, consider the conjunction connective $\wedge$. In order for $A \wedge B$ to be a proposition, we need a way to check its truth. Since $A \wedge B$ is intended to be true whenever both A and B are true, we use the following inference rule to explain $A \wedge B$ as a proposition; we assume that both A and B are propositions, and abbreviate a truth judgment "A is true" as $A\ \mathit{true}$:

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge\mathsf{I}$$

The rule $\wedge\mathsf{I}$ says that if A is true and B is true, then $A \wedge B$ is true. It follows the usual interpretation of an inference rule: if the premises hold, then the conclusion holds. We use the rule $\wedge\mathsf{I}$ to construct a proof $\mathcal{D}$ of $A \wedge B\ \mathit{true}$ from a proof $\mathcal{D}_A$ of $A\ \mathit{true}$ and a proof $\mathcal{D}_B$ of $B\ \mathit{true}$; we write $\begin{array}{c}\mathcal{D}_A\\ A\ \mathit{true}\end{array}$ to mean that $\mathcal{D}_A$ is a proof of $A\ \mathit{true}$:

$$\mathcal{D} \quad = \quad \frac{\begin{array}{c}\mathcal{D}_A\\ A\ \mathit{true}\end{array} \quad \begin{array}{c}\mathcal{D}_B\\ B\ \mathit{true}\end{array}}{A \wedge B\ \mathit{true}}\ \wedge\mathsf{I}$$

Thus $A \wedge B$ is a proposition because we can check its truth according to the rule $\wedge\mathsf{I}$, whereas $A \wedge B\ \mathit{true}$ is a judgment because we either know it or do not know it, depending on the existence of a proof.

The rule $\wedge\mathsf{I}$ above is called an *introduction rule* for the conjunction connective $\wedge$, since its conclusion deduces a truth judgment with $\wedge$, or introduces $\wedge$. A dual concept is an *elimination rule*, whose premises exploit a truth judgment with $\wedge$ to prove another judgment in the conclusion, or eliminates $\wedge$. In the case of $\wedge$, there are two elimination rules, $\wedge\mathsf{E_L}$ and $\wedge\mathsf{E_R}$:

$$\frac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\ \wedge\mathsf{E_L} \qquad\qquad \frac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\ \wedge\mathsf{E_R}$$

These elimination rules make sense because $A \wedge B\ \mathit{true}$ implies both $A\ \mathit{true}$ and $B\ \mathit{true}$. We will later discuss their properties in a more formal way.

It is important that in a judgmental formulation of logic, the notion of judgment takes priority over the notion of proposition. Specifically the notion of judgment does not depend on propositions, and a new kind of judgment is defined only in terms of existing judgments (but without using existing connectives or modalities). On the other hand, propositions are always explained with existing judgments (including at least truth judgments), and a new connective or modality is defined so as to compactly represent the knowledge expressed by existing judgments. For example, we could define a falsehood judgment $A\ \mathit{false}$ as "$A\ \mathit{true}$ does not hold," and then use a new modality $\neg$ with the following introduction rule:

$$\frac{A\ \mathit{false}}{\neg A\ \mathit{true}}\ \neg\mathsf{I}$$

We say that the rule $\neg\mathsf{I}$ *internalizes* $A\ \mathit{false}$ as a proposition $\neg A$.

If the definition of a connective or modality involves another connective or modality, we say that orthogonality is destroyed in the sense that the two connectives or modalities cannot be developed independently, or orthogonally. In this dissertation, we use no connective or modality destroying orthogonality.

Categorical judgments and hypothetical judgments

A judgment such as "A is true" is called a *categorical judgment* because it involves no hypotheses and is thus unconditional. Another judgment that we need is a *hypothetical judgment*, which involves hypotheses. A general form of hypothetical judgment reads "if judgments $J_1, \cdots, J_n$ hold, then a judgment J holds," written as $J_1, \cdots, J_n \vdash J$. We refer to J_i, $1 \leq i \leq n$, as an *antecedent* and J as the *succedent*.

A hypothetical judgment $J_1, \cdots, J_n \vdash J$ becomes evident by a proof of J in which $J_1, \cdots, J_n$ are assumed to be evident without proofs. Such a proof $\mathcal{D}$ is called a *hypothetical proof* and is written as follows:

$$\mathcal{D} \quad = \quad \begin{array}{ccc} J_1 & \cdots & J_n \\ & \ddots \;\; \iddots & \\ & J & \end{array} \quad \} \text{ inference rules}$$

Inference rules here use judgment J_i without requiring a proof, that is, as a hypothesis. We say that a hypothesis J_i is *discharged* when inference rules use it to deduce J_i. Note that a hypothetical proof of $\cdot \vdash J$ (with no antecedent) is essentially a proof of judgment J and vice versa, since both proofs show that J holds categorically.[3]

The notion of hypothetical proof is illustrated by the implication connective $\supset$. In order for $A \supset B$ to be a proposition, we need a way to check its truth. Since $A \supset B$ is intended to be true whenever $A\ \mathit{true}$ implies $B\ \mathit{true}$, the introduction rule uses a hypothetical proof in its premise:

$$\frac{\begin{array}{c} [A\ \mathit{true}] \\ \vdots \\ B\ \mathit{true} \end{array}}{A \supset B\ \mathit{true}} \supset\mathsf{I}$$

The elimination rule for $\supset$ exploits $A \supset B\ \mathit{true}$ in its premises to prove $B\ \mathit{true}$ in its conclusion:

$$\frac{A \supset B\ \mathit{true} \quad A\ \mathit{true}}{B\ \mathit{true}} \supset\mathsf{E}$$

The rule $\supset\mathsf{E}$ makes sense because $A \supset B\ \mathit{true}$ licenses us to deduce $B\ \mathit{true}$ if $A\ \mathit{true}$ holds, which is the case by the second premise.

Our definition of hypothetical judgments makes two implicit assumptions: 1) the order of antecedents is immaterial; 2) an antecedent may be used zero or more times in a hypothetical proof. These assumptions are formally stated in the three structural rules of hypothetical judgments:

(Exchange)	If $J_1, \cdots, J_i, J_{i+1}, \cdots, J_n \vdash J$, then $J_1, \cdots, J_{i+1}, J_i, \cdots, J_n \vdash J$.
(Weakening)	If $J_1, \cdots, J_n \vdash J$, then $J_1, \cdots, J_n, J_{n+1} \vdash J$ for any judgment J_{n+1}.
(Contraction)	If $J_1, \cdots, J_i, J_i, \cdots, J_n \vdash J$, then $J_1, \cdots, J_i, \cdots, J_n \vdash J$.

A hypothetical proof can be combined with another hypothetical proof. For example, a hypothetical proof $\mathcal{D}$ of $J_1, \cdots, J_n \vdash J$ is combined with a hypothetical proof $\mathcal{E}_1$ of $J_2, \cdots, J_n \vdash J_1$ to produce another hypothetical proof, written as $[\mathcal{E}_1/J_1]\mathcal{D}$, of $J_2, \cdots, J_n \vdash J$:

$$[\mathcal{E}_1/J_1]\mathcal{D} \quad = \quad \begin{array}{ccccc} & J_2 & \cdots & J_n & \\ \mathcal{E}_1 & \vdots & & \vdots & \\ J_1 & J_2 & \cdots & J_n & \\ & \ddots & & \iddots & \} \ \mathcal{D} \\ & & J & & \end{array}$$

[3]This equivalence does not mean that a hypothetical judgment $\cdot \vdash J$ is equivalent to judgment J. While the former states that J holds categorically, the latter is unaware of whether there are hypotheses or not, and could be even a hypothesis in a hypothetical proof. For example, from the assumption that J implies J', we can show that $\cdot \vdash J$ implies $\cdot \vdash J'$. The converse is not the case, however.

Note that hypotheses $J_2, \cdots, J_n$ may be used twice: when proving J_1 in $\mathcal{E}_1$ and when proving J in $\mathcal{D}$. This property of hypothetical judgments that a hypothetical proof can be substituted into another hypothetical proof is called the *substitution principle*:

- (Substitution principle) If $\Gamma \vdash J$ and $\Gamma, J \vdash J'$, then $\Gamma \vdash J'$.

A convenient way to prove hypothetical judgments is to use inference rules for hypothetical judgments without relying on hypothetical proofs. For example, we can explain the implication connective $\supset$ with the following inference rules for hypothetical judgments; we abbreviate a collection of antecedents as Γ:

$$\frac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}{\Gamma \vdash A \supset B\ \mathit{true}} \supset\!\mathsf{I} \qquad \frac{\Gamma \vdash A \supset B\ \mathit{true} \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}} \supset\!\mathsf{E}$$

Here the introduction rule $\supset\!\mathsf{I}$ uses hypothetical judgments to express that a proposition $A \supset B$ is true whenever $A\ \mathit{true}$ implies $B\ \mathit{true}$; the elimination rule $\supset\!\mathsf{E}$ uses hypothetical judgments to express that $A \supset B\ \mathit{true}$ licenses us to deduce $B\ \mathit{true}$ if $A\ \mathit{true}$ holds. A proof of $\Gamma \vdash J$ with these inference rules guarantees the existence of a corresponding hypothetical proof of $\Gamma \vdash J$.

A special form of hypothetical judgment $J_1, \cdots, J_i, \cdots, J_n \vdash J_i$ (where the succedent matches an antecedent) is evident by a vacuous proof. The following inference rule, called the *hypothesis rule*, expresses this property of hypothetical judgments; it simply says that any hypothesis can be used:

$$\frac{}{\Gamma, J \vdash J}\ \mathsf{Hyp}$$

From now on, we assume that antecedents and succedents in hypothetical judgments are all basic judgments. For example, we do not consider such hypothetical judgments as $(\Gamma_1 \vdash J_1) \vdash J_2$ and $\Gamma_1 \vdash (\Gamma_2 \vdash J)$.

The Curry-Howard isomorphism

The Curry-Howard isomorphism connects logic and programming languages by representing a proof of a judgment with a program of a corresponding type. In other words, a well-typed program is a compact representation of a valid proof under the Curry-Howard isomorphism. Typically we apply the Curry-Howard isomorphism by translating inference rules of logic into typing rules of a programming language. By convention, a typing rule is given the same name as the inference rule from which it is derived.

As an example, we consider the logic of truth with the conjunction connective $\wedge$ and the implication connective $\supset$. Under the Curry-Howard isomorphism, the logic corresponds to the type system of the λ-calculus with product types. A proof $\mathcal{D}$ of $A\ \mathit{true}$ is represented with a *proof term* M of type A. Note that A is interpreted both as a proposition and as a type. We use a judgment $M : A$ to mean that proof term M represents a proof of $A\ \mathit{true}$, or that proof term M has type A. Thus we have the following correspondence:

$$\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array} \quad \Leftrightarrow \quad M : A$$

Now consider the use of the inference rule $\wedge\mathsf{I}$ in constructing a proof $\mathcal{D}$ of $A \wedge B\ \mathit{true}$ from a proof $\mathcal{D}_A$ of $A\ \mathit{true}$ and a proof $\mathcal{D}_B$ of $B\ \mathit{true}$. When proof terms M_A and M_B represent $\mathcal{D}_A$ and $\mathcal{D}_B$, respectively, we use a *product term* (M_A, M_B) of product type $A \wedge B$ to represent $\mathcal{D}$. Thus the inference rule $\wedge\mathsf{I}$ is translated into the following typing rule:

$$\frac{M_A : A \quad M_B : B}{(M_A, M_B) : A \wedge B} \wedge\mathsf{I}$$

$$\frac{}{\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}}\ \mathsf{Hyp} \qquad \frac{}{\Gamma, x : A \vdash x : A}\ \mathsf{Hyp}$$

$$\frac{\Gamma \vdash A\ \mathit{true} \quad \Gamma \vdash B\ \mathit{true}}{\Gamma \vdash A \wedge B\ \mathit{true}}\ \wedge\mathsf{I} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \wedge B}\ \wedge\mathsf{I}$$

$$\frac{\Gamma \vdash A \wedge B\ \mathit{true}}{\Gamma \vdash A\ \mathit{true}}\ \wedge\mathsf{E_L} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{fst}\ M : A}\ \wedge\mathsf{E_L}$$

$$\frac{\Gamma \vdash A \wedge B\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}}\ \wedge\mathsf{E_R} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{snd}\ M : B}\ \wedge\mathsf{E_R}$$

$$\frac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}{\Gamma \vdash A \supset B\ \mathit{true}}\ \supset\mathsf{I} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \supset B}\ \supset\mathsf{I}$$

$$\frac{\Gamma \vdash A \supset B\ \mathit{true} \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}}\ \supset\mathsf{E} \qquad \frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \supset\mathsf{E}$$

Figure 2.1: Translation of inference rules for hypothetical judgments into typing rules.

We use *projection terms* $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$ in translating the rules $\wedge\mathsf{E_L}$ and $\wedge\mathsf{E_R}$:

$$\frac{M : A \wedge B}{\mathsf{fst}\ M : A}\ \wedge\mathsf{E_L} \qquad \frac{M : A \wedge B}{\mathsf{snd}\ M : B}\ \wedge\mathsf{E_R}$$

When a hypothetical proof uses $A\ \mathit{true}$ as a hypothesis, it assumes the existence of a proof. Since such a proof is actually unknown, it cannot be represented with a concrete proof term. Hence it is represented with a *variable* x, a special proof term which can be replaced by another proof term. Then a proof $\mathcal{D}$ of $A_1\ \mathit{true}, \cdots, A_n\ \mathit{true} \vdash A\ \mathit{true}$ is represented with a proof term M satisfying a hypothetical judgment $x_1 : A_1, \cdots, x_n : A_n \vdash M : A$, which means that proof term M has type A under the assumption that variable x_i, $1 \leq i \leq n$, has type A_i:

$$\overset{\mathcal{D}}{A_1\ \mathit{true}, \cdots, A_n\ \mathit{true} \vdash A\ \mathit{true}} \qquad \Leftrightarrow \qquad x_1 : A_1, \cdots, x_n : A_n \vdash M : A$$

We refer to a collection of judgments $x_1 : A_1, \cdots, x_n : A_n$ as a *typing context*. As with collections of antecedents, we abbreviate typing contexts as Γ; all variables in a typing context are assumed to be distinct.

With the correspondence of hypothetical judgments above, inference rules for hypothetical judgments in logic are translated into typing rules for hypothetical judgments $\Gamma \vdash M : A$. For example, the inference rules $\supset\mathsf{I}$ and $\supset\mathsf{E}$ are translated into the following typing rules, which use a *lambda abstraction* $\lambda x{:}A.\, M$ and a *lambda application* $M\ N$ as proof terms:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \supset B}\ \supset\mathsf{I} \qquad \frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \supset\mathsf{E}$$

Figure 2.1 shows inference rules for hypothetical judgments in logic (shown in the left column) and their translation into typing rules (shown in the right column). The left column shows inference rules for hypothetical judgments, and right column shows corresponding typing rules. The hypothesis rule Hyp is translated into a typing rule, also called the hypothesis rule, that typechecks a variable. The typing rules in the right column constitute the type system of the λ-calculus with product types.

As a hypothetical proof can be substituted into another hypothetical proof, a proof term can also be substituted into another proof term. Suppose $\Gamma \vdash M : A$ and $\Gamma, x : A \vdash N : B$. M and N represent hypothetical proofs $\mathcal{D}$ and $\mathcal{E}$ of $\Gamma \vdash A\ \mathit{true}$ and $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$, respectively, where we use the same symbol Γ for the collection of antecedents corresponding to the typing context Γ. If we replace all occurrences of x in N by M, we obtain a proof term, written as $[M/x]N$, which contains no occurrence of x. The substitution principle for proof terms states that $[M/x]N$ represents the hypothetical proof $[\mathcal{D}/A\ \mathit{true}]\mathcal{E}$ of $\Gamma \vdash B\ \mathit{true}$:

- (Substitution principle)
  If $\Gamma \vdash M : A$ and $\Gamma, x : A \vdash N : B$, then $\Gamma \vdash [M/x]N : B$.

$A\ \mathit{true}$ and $\Gamma \vdash A\ \mathit{true}$ are called *synthetic judgments* because no prior information on their proofs is given and we search for, or *synthesize*, their proofs from inference rules. In contrast, $M : A$ and $\Gamma \vdash M : A$ are called *analytic judgments* because their proofs are already represented in M and can be reconstructed by *analyzing* M. To prove $M : A$ or $\Gamma \vdash M : A$ with typing rules, we only have to analyze M because it determines which typing rule should be applied to deduce $M : A$ or $\Gamma \vdash M : A$. For example, if M is a product term (*i.e.*, $M = (M_1, M_2)$), a deduction of $\Gamma \vdash M : A$ always ends with an application of the typing rule $\wedge\mathsf{I}$. For this reason, a deduction of $M : A$ or $\Gamma \vdash M : A$ is often called a *derivation* rather than a proof.

When we construct a (unique) derivation of $M : A$ or $\Gamma \vdash M : A$, we check if M indeed represents a proof of $A\ \mathit{true}$, rather than searching for a yet unknown proof. Such a derivation effectively typechecks M by testing if M indeed has type A, and we call $M : A$ and $\Gamma \vdash M : A$ *typing judgments*.

Reduction and expansion rules

All the inference rules presented so far make sense intuitively, but their correctness is yet to be established in a formal way. To this end, we show that the inference rules satisfy two properties: *local soundness* and *local completeness*. Under the Curry-Howard isomorphism, the two properties correspond to reduction and expansion rules for proof terms, thus culminating in a foundation for operational semantics of programming languages.

An introduction rule compresses the knowledge expressed in its premises into a truth judgment in the conclusion, whereas an elimination rule retrieves the knowledge compressed within a truth judgment in a premise to deduce another judgment in the conclusion. The local soundness property states that the knowledge retrieved from a judgment by an elimination rule is only part of the knowledge compressed within that judgment. Therefore, if the local soundness property fails, the elimination rule is too strong in the sense that it is capable of contriving some knowledge that cannot be justified by that judgment. The local completeness property states that the knowledge retrieved from a judgment by an elimination rule includes at least the knowledge compressed within that judgment. Therefore, if the local completeness property fails, the elimination rule is too weak in the sense that it is incapable of retrieving all the knowledge compressed within that judgment. If an elimination rule satisfies both properties, it retrieves exactly the same knowledge compressed within a judgment in a premise.

We verify the local soundness property by showing how to reduce a proof in which an introduction rule is immediately followed by a corresponding elimination rule. As an example, consider the following proof for the conjunction connective $\wedge$:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{A\ \mathit{true}} \quad \overset{\mathcal{E}}{B\ \mathit{true}}}{A \wedge B\ \mathit{true}}\,\wedge\mathsf{I}}{A\ \mathit{true}}\,\wedge\mathsf{E_L}$$

The elimination rule $\wedge\mathsf{E_L}$ is not too strong because what it deduces in the conclusion, namely $A\ \mathit{true}$, is one of the two judgments used to deduce $A \wedge B\ \mathit{true}$. Hence the whole proof reduces to a simpler proof $\mathcal{D}$:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{A\ \mathit{true}} \quad \overset{\mathcal{E}}{B\ \mathit{true}}}{A \wedge B\ \mathit{true}}\,\wedge\mathsf{I}}{A\ \mathit{true}}\,\wedge\mathsf{E_L} \qquad \Longrightarrow_R \qquad \overset{\mathcal{D}}{A\ \mathit{true}}$$

If the elimination rule was too strong (*e.g.*, deducing $A \supset B\ \mathit{true}$ somehow), the proof would not be reducible. As another example, consider the proof for the implication connective $\supset$:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}}{\Gamma \vdash A \supset B\ \mathit{true}} \supset\mathsf{I} \quad \overset{\mathcal{E}}{\Gamma \vdash A\ \mathit{true}}}{\Gamma \vdash B\ \mathit{true}} \supset\mathsf{E}$$

By the substitution principle, the whole proof reduces to a simpler proof $[\mathcal{E}/A\ \mathit{true}]\mathcal{D}$:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}}{\Gamma \vdash A \supset B\ \mathit{true}} \supset\mathsf{I} \quad \overset{\mathcal{E}}{\Gamma \vdash A\ \mathit{true}}}{\Gamma \vdash B\ \mathit{true}} \supset\mathsf{E} \qquad \Longrightarrow_R \qquad \begin{array}{c}[\mathcal{E}/A\ \mathit{true}]\mathcal{D} \\ \Gamma \vdash B\ \mathit{true}\end{array}$$

We refer to these reductions $\Longrightarrow_R$ as *local reductions*.

We verify the local completeness property by showing how to expand a proof of a judgment into another proof in which one or more elimination rules are followed by an introduction rule for the same judgment. As an example, consider a proof $\mathcal{D}$ of $A \wedge B\ \mathit{true}$. The elimination rules $\wedge\mathsf{E_L}$ and $\wedge\mathsf{E_R}$ are not too weak because what they deduce in their conclusions, namely $A\ \mathit{true}$ and $B\ \mathit{true}$, are sufficient to reconstruct another proof of $A \wedge B\ \mathit{true}$:

$$\overset{\mathcal{D}}{A \wedge B\ \mathit{true}} \qquad \Longrightarrow_E \qquad \dfrac{\dfrac{\overset{\mathcal{D}}{A \wedge B\ \mathit{true}}}{A\ \mathit{true}} \wedge\mathsf{E_L} \quad \dfrac{\overset{\mathcal{D}}{A \wedge B\ \mathit{true}}}{B\ \mathit{true}} \wedge\mathsf{E_R}}{A \wedge B\ \mathit{true}} \wedge\mathsf{I}$$

If the elimination rules were too weak (*e.g.*, being unable to deduce $A\ \mathit{true}$ somehow), the proof would not be expandable. As another example, consider a proof $\mathcal{D}$ of $\Gamma \vdash A \supset B\ \mathit{true}$. By the weakening property, $\mathcal{D}$ is also a proof of $\Gamma, A\ \mathit{true} \vdash A \supset B\ \mathit{true}$. Then we can reconstruct another proof of $A \supset B\ \mathit{true}$ by expanding $\mathcal{D}$:

$$\overset{\mathcal{D}}{\Gamma \vdash A \supset B\ \mathit{true}} \qquad \Longrightarrow_E \qquad \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, A\ \mathit{true} \vdash A \supset B\ \mathit{true}} \quad \dfrac{}{\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}}\mathsf{Hyp}}{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}} \supset\mathsf{E}}{\Gamma \vdash A \supset B\ \mathit{true}} \supset\mathsf{I}$$

We refer to these expansions $\Longrightarrow_E$ as *local expansions*.

Since proof terms are essentially proofs, local reductions and expansions induce reduction and expansion rules for proof terms:

$$\begin{array}{ccc} \mathsf{fst}\ (M, N) & \Longrightarrow_R & M \\ \mathsf{snd}\ (M, N) & \Longrightarrow_R & N \\ (\lambda x\!:\!A.\, M)\ N & \Longrightarrow_R & [N/x]M \end{array}$$

$$\begin{array}{ccc} M : A \wedge B & \Longrightarrow_E & (\mathsf{fst}\ M, \mathsf{snd}\ M) \\ M : A \supset B & \Longrightarrow_E & \lambda x\!:\!A.\, M\ x \end{array}$$

Note that these reduction and expansion rules preserve the type of a given proof term. That is, if $M \Longrightarrow_R N$ or $M \Longrightarrow_E N$, then $\Gamma \vdash M : A$ implies $\Gamma \vdash N : A$. The reduction rules are called the β-reduction rules, and the expansion rules are called the η-expansion rules.

In a programming language based upon the λ-calculus, a program is defined as a well-typed closed proof term, that is, a proof term M such that $\cdot \vdash M : A$ for a certain type A. Usually we run a program by applying reduction rules under a specific *reduction strategy*. For example, the *call-by-name* reduction strategy reduces a program $(\lambda x\!:\!A.\, M)\ N$ to $[N/x]M$ (by the β-reduction rule) regardless of the form of term N. In contrast, the *call-by-value* reduction strategy reduces $(\lambda x\!:\!A.\, M)\ N$ to $[N/x]M$ only if no reduction rule is applicable to N (*i.e.*, N is a value). Thus the operational semantics of a programming language based upon the λ-calculus is specified by the reduction strategy for applying reduction rules.

2.2.2 Semantics of modal logic

Modal logic is a form of logic in which truth may be qualified by modalities. Examples of modalities common in the literature are the *necessity* modality $\Box$ and the *possibility* modality $\Diamond$. Informally "$\Box A$ is true" means "A is necessarily true," and "$\Diamond A$ is true" means "A is possibly true." Thus modal logic is more expressive than ordinary logic without modalities, and when applied to the design of a programming language, it enables the type system to specify richer properties that would otherwise be difficult to specify.

One popular way to explain the semantics of modal logic is the possible world interpretation [35, 71]. It assumes a set of worlds and relativizes truth to worlds. That is, instead of ordinary truth "A is true," it uses *relative truth* "A is true at world ω" as the primitive notion. Hence the same proposition may be true at one world but not at another world.

The possible world interpretation also assumes an *accessibility relation* $\leq$ between worlds to explain the meaning of each modality. For example, the necessity and possibility modalities are defined as follows:

- $\Box A$ is true at world ω if for every world ω' accessible from ω (*i.e.*, $\omega \leq \omega'$), A is true at ω'.

- $\Diamond A$ is true at world ω if A is true at some world ω' accessible from ω (*i.e.*, $\omega \leq \omega'$).

Ordinary connectives (such as $\supset$ and $\wedge$) are explained locally at individual worlds, irrespective of $\leq$. For example, $A \supset B$ is true at world ω if "A is true at ω" implies "B is true at ω."

With the above definition of the modalities $\Box$ and $\Diamond$, some proposition becomes true at every world, regardless of the accessibility relation $\leq$. For example, $\Box(A \supset B) \supset (\Box A \supset \Box B)$ is true at every world, since $\Box(A \supset B)$ and $\Box A$ are sufficient to show that B is true at any accessible world. Moreover various systems of modal logic are obtained by requiring $\leq$ to satisfy certain properties. The following table shows some properties of $\leq$ and corresponding propositions that become true at every world:

property of $\leq$		proposition
reflexivity	$\forall\omega.\ \omega \leq \omega$	$\Box A \supset A$
symmetry	$\forall\omega.\forall\omega'.\ \omega \leq \omega'$ implies $\omega' \leq \omega$	$A \supset \Box\Diamond A$
transitivity	$\forall\omega.\forall\omega'.\forall\omega''.\ \omega \leq \omega'$ and $\omega' \leq \omega''$ imply $\omega \leq \omega''$	$\Box A \supset \Box\Box A$
Euclideanness	$\forall\omega.\forall\omega'.\forall\omega''.\ \omega \leq \omega'$ and $\omega \leq \omega''$ imply $\omega' \leq \omega''$	$\Diamond A \supset \Box\Diamond A$

For example, if $\leq$ is reflexive and transitive, we obtain a system of modal logic, usually referred to as S4, in which both $\Box A \supset A$ and $\Box A \supset \Box\Box A$ are true at every world.

The semantics of modal logic can also be explained without explicitly using the notion of world [62, 8, 60]. In their judgmental formulation of modal logic, Pfenning and Davies [60] define a *validity judgment* $A\ valid$ as $\cdot \vdash A\ true$, and internalize $A\ valid$ as a modal proposition $\Box A$:

$$\frac{A\ valid}{\Box A\ true}\ \Box\mathsf{I}$$

Thus $\Box A\ true$ is interpreted as A being true at a world about which we know nothing, or equivalently, at every world. (Note that a judgment is defined first and then a corresponding modality is introduced.) A *possibility judgment* $A\ poss$ is based upon the interpretation of $A\ poss$ as A being true at a certain world, but still its definition does not use worlds explicitly:

1. If $\Gamma \vdash A\ true$, then $\Gamma \vdash A\ poss$.

2. If $\Gamma \vdash A\ poss$ and $A\ true \vdash B\ poss$, then $\Gamma \vdash B\ poss$.

A possibility judgment $A\ poss$ is internalized as a modal proposition $\Diamond A$:

$$\frac{A\ poss}{\Diamond A\ true}\ \Diamond\mathsf{I}$$

The possible world interpretation is richer than the judgmental formulation in that some proposition is true in the possible world interpretation but not in the judgmental formulation. An example of such a proposition is $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$. It is true in the possible world interpretation as follows; we write $A \mathbin{@} \omega$ for A being true at world ω:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Diamond A \supset \Box B \mathbin{@} \omega, A \mathbin{@} \omega' \vdash \Diamond A \supset \Box B \mathbin{@} \omega}\ \mathsf{Hyp} \quad \dfrac{\dfrac{}{\Diamond A \supset \Box B \mathbin{@} \omega, A \mathbin{@} \omega' \vdash A \mathbin{@} \omega'}\ \mathsf{Hyp}}{\Diamond A \supset \Box B \mathbin{@} \omega, A \mathbin{@} \omega' \vdash \Diamond A \mathbin{@} \omega}\ \Diamond\mathsf{I}}{\Diamond A \supset \Box B \mathbin{@} \omega, A \mathbin{@} \omega' \vdash B \mathbin{@} \omega'}\ {\supset}\mathsf{E}}{\omega \leq \omega',\ \Diamond A \supset \Box B \mathbin{@} \omega \vdash A \supset B \mathbin{@} \omega'}\ {\supset}\mathsf{I}}{\Diamond A \supset \Box B \mathbin{@} \omega \vdash \Box(A \supset B) \mathbin{@} \omega}\ \Box\mathsf{I}}{\cdot \vdash (\Diamond A \supset \Box B) \supset \Box(A \supset B) \mathbin{@} \omega}\ {\supset}\mathsf{I}$$

Its truth is, however, not provable in the judgmental formulation:

$$\dfrac{\dfrac{\dfrac{???}{\cdot \vdash A \supset B\ true}}{\Diamond A \supset \Box B\ true \vdash \Box(A \supset B)\ true}\ \Box\mathsf{I}}{\cdot \vdash (\Diamond A \supset \Box B) \supset \Box(A \supset B)\ true}\ {\supset}\mathsf{I}$$

In a certain sense, the possible world interpretation is inherently more expressive than the judgmental formulation because it explicitly specifies the world at which a proposition is true. On the other hand, it may not be a good basis for the type system of a programming language, since the use of the accessibility relation in proofs implies that the type system also needs to reason about the relation between worlds, which can be difficult depending on the concrete notion of world chosen by the type system. The judgmental formulation lends itself well to this purpose because it does not use worlds explicitly in the inference rules.

The logic for $\lambda_{\bigcirc}$ combines the possible world interpretation and the judgmental style by assuming an accessibility relation between worlds and relativizing all judgments to worlds. For example, it uses a truth judgment of the form $A\ true \mathbin{@} \omega$ to mean that A is true at world ω. Its inference rules, however, do not use judgments showing accessibility between two worlds, as is the case in the judgmental formulation of modal logic (see Simpson [71] for a system of modal logic which uses such judgments in inference rules). Instead it requires the accessibility relation to satisfy a certain condition (monotonicity), which eliminates the need for such judgments in inference rules. Since the possible world interpretation in $\lambda_{\bigcirc}$ is to use the same worlds that are part of the run-time system, lack of such judgments in inference rules implies that the type system of $\lambda_{\bigcirc}$ does not explicitly model changes in the run-time system, as is the case in a typical type system.

2.3 Language $\lambda_{\bigcirc}$

Pfenning and Davies [60] present a monadic language which reformulates Moggi's monadic metalanguage λ_{ml} [44, 45]. It applies the Curry-Howard isomorphism to lax logic formulated in the judgmental style (with a lax truth judgment $A\ lax$):

1. If $\Gamma \vdash A\ true$, then $\Gamma \vdash A\ lax$.

2. If $\Gamma \vdash A\ \mathit{lax}$ and $\Gamma, A\ \mathit{true} \vdash B\ \mathit{lax}$, then $\Gamma \vdash B\ \mathit{lax}$.

$\lambda_{\bigcirc}$ is essentially the monadic language of Pfenning and Davies coalesced with the possible world interpretation. The difference is that in $\lambda_{\bigcirc}$, the definition of each judgment relies only on truth and the accessibility relation, instead of clauses describing its properties (such as the above two clauses). In other words, the definition of each judgment directly conveys its intuitive meaning.

2.3.1 Logic for $\lambda_{\bigcirc}$

The development of $\lambda_{\bigcirc}$ begins by formulating the logic for $\lambda_{\bigcirc}$. Since the logic for $\lambda_{\bigcirc}$ uses the possible world interpretation, we first define an accessibility relation $\leq$ between worlds. Now a world refers to the same notion that describes part of the run-time system.

Definition 2.1. *A world ω' is accessible from another world ω, written as $\omega \leq \omega'$, if there exists a world effect that causes a transition from ω to ω'.*

As it describes transitions between worlds when world effects are produced, the accessibility relation $\leq$ is a *temporal* relation between worlds. If $\omega \leq \omega'$, we say that ω' is a future world of ω and that ω is a past world of ω'. Note that $\leq$ is reflexive and transitive, since a vacuous world effect causes a transition to the same world and the combination of two world effects can be regarded as a single world effect.

The logic for $\lambda_{\bigcirc}$ uses two kinds of basic judgments, both of which are relativized to worlds:

- A *truth judgment* $A\ \mathit{true} \ @\ \omega$ means that A is true at world ω.
- A *computability judgment* $A\ \mathit{comp} \ @\ \omega$ means that A is true at some future world of ω, that is, $A\ \mathit{true} \ @\ \omega'$ holds where $\omega \leq \omega'$.

A truth judgment $A\ \mathit{true} \ @\ \omega$ represents a known fact about world ω. Since a future world can be reached only by producing some world effect, a computability judgment $A\ \mathit{comp} \ @\ \omega$ may be interpreted as meaning that A becomes true after producing some world effect at world ω.

The following properties of hypothetical judgments characterize truth judgments, where J is either a truth judgment or a computability judgment:

Characterization of truth judgments

1. $\Gamma, A\ \mathit{true} \ @\ \omega \vdash A\ \mathit{true} \ @\ \omega$.
2. If $\Gamma \vdash A\ \mathit{true} \ @\ \omega$ and $\Gamma, A\ \mathit{true} \ @\ \omega \vdash J$, then $\Gamma \vdash J$.

The first clause expresses that $A\ \mathit{true} \ @\ \omega$ may be used as a hypothesis. The second clause expresses the substitution principle for truth judgments.

The definition of computability judgments gives the following characterization, which is an adaptation of the characterization of lax truth for the possible world interpretation:

Characterization of computability judgments

1. If $\Gamma \vdash A\ \mathit{true} \ @\ \omega$, then $\Gamma \vdash A\ \mathit{comp} \ @\ \omega$.
2. If $\Gamma \vdash A\ \mathit{comp} \ @\ \omega$ and $\Gamma, A\ \mathit{true} \ @\ \omega' \vdash B\ \mathit{comp} \ @\ \omega'$ for any world ω' such that $\omega \leq \omega'$, then $\Gamma \vdash B\ \mathit{comp} \ @\ \omega$.

The first clause expresses that if A is true at ω, then A becomes true without producing any world effect at ω. It follows from the reflexivity of $\leq$: if A *true* $@\ \omega$ holds, then A is true at ω, which is accessible from ω itself, and hence A *comp* $@\ \omega$ holds. The second clause expresses that if A is true at ω' after producing some world effect at ω, we may use A *true* $@\ \omega'$ as a hypothesis in deducing a judgment at ω'. If the judgment at ω' is a computability judgment B *comp* $@\ \omega'$, the transitivity of $\leq$ allows us to deduce B *comp* $@\ \omega$:

Proof of the second clause. Assume that A *comp* $@\ \omega$ implies A *true* $@\ \omega_1$ where $\omega \leq \omega_1$. We prove B *comp* $@\ \omega$ from hypotheses Γ as follows:

A *comp* $@\ \omega$ holds because $\Gamma \vdash A$ *comp* $@\ \omega$.
A *true* $@\ \omega_1$ holds by the assumption on A *comp* $@\ \omega$.
B *comp* $@\ \omega_1$ holds because Γ, A *true* $@\ \omega_1 \vdash B$ *comp* $@\ \omega_1$.
B *true* $@\ \omega_2$ holds for some world ω_2 such that $\omega_1 \leq \omega_2$ (by the definition of B *comp* $@\ \omega_1$).
B *comp* $@\ \omega$ holds because $\omega \leq \omega_2$ by the transitivity of $\leq$ (*i.e.*, $\omega \leq \omega_1 \leq \omega_2$). $\square$

We use the second clause as the substitution principle for computability judgments.

Monotonicity of the accessibility relation $\leq$

We intend to use world effects for accumulating more knowledge, but not for discarding existing knowledge. Informally a world effect causes a transition to a world where more facts are known and more world effects can be produced. The monotonicity of the accessibility relation $\leq$ formalizes our intention to use world effects only for accumulating more knowledge:

Definition 2.2. *The accessibility relation $\leq$ is* monotonic *if for two worlds ω and ω' such that $\omega \leq \omega'$,*
1) A true $@\ \omega$ implies A true $@\ \omega'$;
2) A_1 true $@\ \omega, \cdots, A_n$ true $@\ \omega \vdash A$ comp $@\ \omega$ implies A_1 true $@\ \omega', \cdots, A_n$ true $@\ \omega' \vdash A$ comp $@\ \omega'$.

The first condition, *monotonicity of truth*, states that a future world inherits all facts known about its past worlds. It proves two new properties of hypothetical judgments:

1. If $\Gamma \vdash A$ *true* $@\ \omega$ and $\omega \leq \omega'$, then $\Gamma \vdash A$ *true* $@\ \omega'$.
2. If Γ, A *true* $@\ \omega' \vdash J$ and $\omega \leq \omega'$, then Γ, A *true* $@\ \omega \vdash J$.

The second condition, *persistence of computation*, states that a world effect that can be produced at world ω under some facts (about ω) can be reproduced at any future world ω' under equivalent facts (about ω'). Unlike monotonicity of truth, it uses hypothetical judgments in which all antecedents are truth judgments at the same world as the succedent. The reason is that a world effect may require some facts about the world at which it is produced (*e.g.*, allocating a new reference requires an argument for initializing a new heap cell), and its corresponding computability judgments at different worlds can be compared for persistence only under equivalent facts about individual worlds.

Note that monotonicity of truth does not imply persistence of computation. For example, if A *comp* $@\ \omega$ holds because A *true* $@\ \omega'$ where $\omega \leq \omega'$, monotonicity of truth allows us to conclude A *comp* $@\ \omega''$ for every world ω'' accessible from ω', but not for every world accessible from ω.

Simplified form of hypothetical judgment

In principle, a hypothetical judgment $\Gamma \vdash J$ imposes no restriction on antecedents Γ and succedent J. That is, if J is a judgment at world ω, then Γ may include both truth judgments and computability judgments at world ω itself, past worlds of ω, future worlds of ω, or even those worlds unrelated to ω. Thus such a general form of hypothetical judgment allows us to express reasoning about not only the present but also the past and future.

Examples of reasoning about the past and future are:

- If there has been a transaction failure in a database system in the *past*, we create a log file *now*.
- If the program has produced no output yet, we stop taking input.
- If the heap cell is deallocated in the *future* and becomes no longer available, we make a copy of it *now*.
- If the program is to open the file eventually, we do not close it.

Since we intend to use $\lambda_{\bigcirc}$ only to reason about the present, the logic for $\lambda_{\bigcirc}$ imposes restrictions on antecedents in hypothetical judgments and uses a simplified form of hypothetical judgment as described below.

First the simplified form uses as antecedents only truth judgments. If a computability judgment is to be exploited, we use as an antecedent a truth judgment that it asserts, as shown in the second clause of the characterization of computability judgments. Second the simplified form uses only judgments at the same world. In other words, a hypothetical proof reasons about one present world and does not consider its relation to past and future worlds (or unrelated worlds). The rationale for the second simplification is two-fold:

1. Facts about past worlds automatically become facts about the present world by the monotonicity of $\leq$. Therefore there is no reason to consider facts about the past.
2. In general, facts about future worlds are unknown to the present world because of the temporal nature of $\leq$. If we were to support reasoning about future worlds, the necessity and possibility modalities would be necessary.

Thus the logic for $\lambda_{\bigcirc}$ uses the following two forms of hypothetical judgments:

- $A_1\ \mathit{true} \ @\ \omega, \cdots, A_n\ \mathit{true} \ @\ \omega \vdash A\ \mathit{true} \ @\ \omega$,
  which is abbreviated as $A_1\ \mathit{true}, \cdots, A_n\ \mathit{true} \vdash_{\mathsf{s}} A\ \mathit{true} \ @\ \omega$.
- $A_1\ \mathit{true} \ @\ \omega, \cdots, A_n\ \mathit{true} \ @\ \omega \vdash A\ \mathit{comp} \ @\ \omega$,
  which is abbreviated as $A_1\ \mathit{true}, \cdots, A_n\ \mathit{true} \vdash_{\mathsf{s}} A\ \mathit{comp} \ @\ \omega$.

As the logic for $\lambda_{\bigcirc}$ requires only the simplified form of hypothetical judgment, we simplify the characterization of truth and computability judgments accordingly. The new characterization of truth judgments is just a special case of the previous characterization:

Characterization of truth judgments with $\Gamma \vdash_{\mathsf{s}} J$

1. $\Gamma, A\ \mathit{true} \vdash_{\mathsf{s}} A\ \mathit{true} \ @\ \omega$.
2. If $\Gamma \vdash_{\mathsf{s}} A\ \mathit{true} \ @\ \omega$ and $\Gamma, A\ \mathit{true} \vdash_{\mathsf{s}} J$, then $\Gamma \vdash J$, where J is a judgment at world ω.

The new characterization of computability judgments does not consider transitions between worlds:

Characterization of computability judgments with $\Gamma \vdash_{\mathsf{s}} J$

1. If $\Gamma \vdash_{\mathsf{s}} A\ \mathit{true} \ @\ \omega$, then $\Gamma \vdash_{\mathsf{s}} A\ \mathit{comp} \ @\ \omega$.
2. If $\Gamma \vdash_{\mathsf{s}} A\ \mathit{comp} \ @\ \omega$ and $\Gamma, A\ \mathit{true} \vdash_{\mathsf{s}} B\ \mathit{comp} \ @\ \omega$, then $\Gamma \vdash_{\mathsf{s}} B\ \mathit{comp} \ @\ \omega$.

Proof of the second clause. Given $\Gamma = A_1\ \mathit{true}, \cdots, A_n\ \mathit{true}$, we write $\Gamma\ @\ \omega$ for $A_1\ \mathit{true}\ @\ \omega, \cdots, A_n\ \mathit{true}\ @\ \omega$. Assume $\Gamma\ @\ \omega \vdash A\ \mathit{comp}\ @\ \omega$ and $\Gamma\ @\ \omega, A\ \mathit{true}\ @\ \omega \vdash B\ \mathit{comp}\ @\ \omega$. For any world ω' such that $\omega \leq \omega'$,

$\Gamma\ @\ \omega', A\ \mathit{true}\ @\ \omega' \vdash B\ \mathit{comp}\ @\ \omega'$ holds by persistence of computation;

$\Gamma\ @\ \omega, A\ \mathit{true}\ @\ \omega' \vdash B\ \mathit{comp}\ @\ \omega'$ holds by monotonicity of truth.

Then $\Gamma\ @\ \omega \vdash B\ \mathit{comp}\ @\ \omega$, or $\Gamma \vdash_{\mathsf{s}} B\ \mathit{comp}\ @\ \omega$, holds by the substitution principle for computability judgments. □

Note that in the second clause, $A\ \mathit{comp}\ @\ \omega$ leads to (as a new hypothesis) a truth judgment at the same world instead of a future world. That is, even if $A\ \mathit{comp}\ @\ \omega$ holds because $A\ \mathit{true}\ @\ \omega'$ where $\omega \leq \omega'$, we use as a new hypothesis $A\ \mathit{true}\ @\ \omega$ instead of $A\ \mathit{true}\ @\ \omega'$. Thus we reason as if the world effect corresponding to $A\ \mathit{comp}\ @\ \omega$ did not cause a transition to the future world ω'. By virtue of the monotonicity of $\leq$, this reasoning provides a simple way to test $B\ \mathit{comp}\ @\ \omega''$ for *every* future world ω'' of ω, as in the previous characterization of computability judgments. The second clause allows the type system of $\lambda_{\bigcirc}$ to typecheck a program producing a sequence of world effects without actually producing them, as will be seen in the next subsection.

2.3.2 Language constructs of $\lambda_{\bigcirc}$

To represent proofs of judgments, we use two syntactic categories: *terms* M, N for truth judgments and *expressions* E, F for computability judgments. Thus the Curry-Howard isomorphism gives the following correspondence, where typing judgments are annotated with worlds where terms or expressions reside:

$$\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\ @\ \omega\end{array} \quad\Leftrightarrow\quad M : A\ @\ \omega \qquad\qquad \begin{array}{c}\mathcal{E}\\ A\ \mathit{comp}\ @\ \omega\end{array} \quad\Leftrightarrow\quad E \div A\ @\ \omega$$

That is, we represent a proof $\mathcal{D}$ of $A\ \mathit{true}\ @\ \omega$ as a term M of type A at world ω, written as $M : A\ @\ \omega$, and a proof $\mathcal{E}$ of $A\ \mathit{comp}\ @\ \omega$ as an expression E of type A at world ω, written as $E \div A\ @\ \omega$. Analogously hypothetical judgments (of the form $\Gamma \vdash_{\mathsf{s}} J$) correspond to typing judgments with typing contexts:

$$\Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega \qquad\qquad \Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega$$

A typing context Γ is a set of bindings $x : A$:

$$\text{typing context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

$x : A$ in Γ means that variable x assumes a term that has type A at a given world (*i.e.*, world ω in $\Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega$ or $\Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega$) but may not typecheck at other worlds. Then a term typing judgment $\Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega$ means that M has type A at world ω if Γ is satisfied at the same world; similarly an expression typing judgment $\Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega$ means that E has type A at world ω if Γ is satisfied at the same world. Alternatively we may think of $\Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega$ or $\Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega$ as typing judgments indexed by worlds.

Terms and expressions form separate sublanguages of $\lambda_{\bigcirc}$. Their difference is manifest in the operational semantics of $\lambda_{\bigcirc}$, which draws a distinction between *evaluations* of terms, involving no worlds, and *computations* of expressions, involving transitions between worlds:

$$M \rightharpoonup V \qquad\qquad E\ @\ \omega \rightharpoondown V\ @\ \omega'$$

A term evaluation $M \rightharpoonup V$ does not interact with the world where term M resides; hence the resultant value V resides at the same world. In contrast, an expression computation $E\ @\ \omega \rightharpoondown V\ @\ \omega'$ may interact

type	A, B	$::=$	$A \supset A \mid \bigcirc A$
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\,M \mid M\ M \mid \mathsf{cmp}\ E$
expression	E, F	$::=$	$M \mid \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$
value	V	$::=$	$\lambda x\!:\!A.\,M \mid \mathsf{cmp}\ E$

Figure 2.2: Abstract syntax for $\lambda_{\bigcirc}$.

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{s}} x : A \mathbin{@} \omega}\ \mathsf{Hyp} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{s}} M : B \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} \lambda x\!:\!A.\,M : A \supset B \mathbin{@} \omega}\ {\supset}\mathsf{I}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M_1 : A \supset B \mathbin{@} \omega \quad \Gamma \vdash_{\mathsf{s}} M_2 : A \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} M_1\ M_2 : B \mathbin{@} \omega}\ {\supset}\mathsf{E} \qquad \frac{\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{cmp}\ E : \bigcirc A \mathbin{@} \omega}\ \bigcirc\mathsf{I}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} M \div A \mathbin{@} \omega}\ \mathsf{Term} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : \bigcirc A \mathbin{@} \omega \quad \Gamma, x : A \vdash_{\mathsf{s}} E \div B \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E \div B \mathbin{@} \omega}\ \bigcirc\mathsf{E}$$

Figure 2.3: Typing rules of $\lambda_{\bigcirc}$.

with world ω where expression E resides, causing a transition to another world ω'; hence the resultant value V may not reside at the same world. Thus term evaluations are always effect-free whereas expression computations are potentially effectful (with respect to world effects).

Note that worlds are required by both the type system and the operational semantics of $\lambda_{\bigcirc}$. That is, worlds are both compile-time objects and run-time objects in the definition of $\lambda_{\bigcirc}$. As worlds are involved in expression computations and hence definitely serve as run-time objects, one could argue that abstractions of worlds rather than worlds themselves (*e.g.*, store typing contexts rather than stores) are more appropriate for the type system. Our view is that worlds are acceptable to use in the type system for the same reason that terms and expressions appear in both the type system and the operational semantics: the type system determines static properties of terms and expressions, and the operational semantics describes how to reduce terms and expressions; likewise the type system determines static properties of worlds (with respect to terms and expressions), and the operational semantics describes transitions between worlds.

Incidentally the type system of $\lambda_{\bigcirc}$ is designed in such a way that only an initial world at which the run-time system starts (*e.g.*, an empty store) is required for typechecking any program. Hence no practical problem arises in implementing the type system as we can simply disregard worlds.

Below we introduce all term and expression constructs of $\lambda_{\bigcirc}$. Figure 2.2 summarizes the abstract syntax for $\lambda_{\bigcirc}$. Figure 2.3 summarizes the typing rules of $\lambda_{\bigcirc}$. We use x, y, z for variables.

Term constructs

As terms represent proofs of truth judgments, the characterization of truth judgments gives properties of terms when interpreted via the Curry-Howard isomorphism. The first clause gives the following rule where variable x is used as a term:

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{s}} x : A \mathbin{@} \omega}\ \mathsf{Hyp}$$

The second clause gives the substitution principle for terms:

Substitution principle for terms

If $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} N : B \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [M/x]N : B \mathbin{@} \omega$.
If $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} E \div B \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [M/x]E \div B \mathbin{@} \omega$.

$[M/x]N$ and $[M/x]E$ denote capture-avoiding *term substitutions* which substitute M for all occurrences of x in N and E. We will give the definition of term substitution after introducing all term and expression constructs.

We apply the Curry-Howard isomorphism to truth judgments by introducing an implication connective $\supset$ such that $\Gamma \vdash_{\mathsf{s}} A \supset B\ \mathit{true} \ @\ \omega$ expresses $\Gamma, A\ \mathit{true} \vdash_{\mathsf{s}} B\ \mathit{true}\ @\ \omega$. It gives the following introduction and elimination rules, where we use a lambda abstraction $\lambda x\!:\!A.\,M$ and a lambda application $M_1\ M_2$ as terms:

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} M : B\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} \lambda x\!:\!A.\,M : A \supset B\ @\ \omega}\ \supset\!\mathsf{I} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M_1 : A \supset B\ @\ \omega \quad \Gamma \vdash_{\mathsf{s}} M_2 : A\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} M_1\ M_2 : B\ @\ \omega}\ \supset\!\mathsf{E}$$

We use a reduction relation $\Rightarrow_{\beta\mathsf{term}}$ in both the term reduction rule for $\supset$ and its corresponding proof reduction:

$$(\lambda x\!:\!A.\,N)\ M \ \Rightarrow_{\beta\mathsf{term}}\ [M/x]N \qquad (\beta_{\supset})$$

$$\frac{\dfrac{\Gamma, x : A \vdash_{\mathsf{s}} N : B\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} \lambda x\!:\!A.\,N : A \supset B\ @\ \omega}\ \supset\!\mathsf{I} \qquad \Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} (\lambda x\!:\!A.\,N)\ M : B\ @\ \omega}\ \supset\!\mathsf{E} \quad \Rightarrow_{\beta\mathsf{term}} \quad \Gamma \vdash_{\mathsf{s}} [M/x]N : B\ @\ \omega$$

Expression constructs

Similarly to truth judgments, we begin by interpreting the characterization of computability judgments in terms of typing judgments. The first clause means that a term of type A is also an expression of the same type:

$$\frac{\Gamma \vdash_{\mathsf{s}} M : A\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} M \div A\ @\ \omega}\ \mathsf{Term}$$

The second clause gives the substitution principle for expressions:

Substitution principle for expressions
If $\Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} F \div B\ @\ \omega$, *then* $\Gamma \vdash_{\mathsf{s}} \langle E/x\rangle F \div B\ @\ \omega$.

Unlike a term substitution $[M/x]F$ which analyzes the structure of F, an *expression substitution* $\langle E/x\rangle F$ analyzes the structure of E instead of F. This is because $\langle E/x\rangle F$ is intended to ensure that both E and F are computed exactly once and in that order: first we compute E to obtain a value; then we proceed to compute F with x bound to the value. Therefore we should not replicate E within F (at those places where x occurs), which would result in computing E multiple times. Instead we should conceptually replicate F within E (at those places where the computation of E finishes) so that the whole computation ends up computing both E and F only once. In this sense, an expression substitution $\langle E/x\rangle F$ substitutes not E into F, but F into E. We will give the definition of expression substitution after introducing all expression constructs.

We apply the Curry-Howard isomorphism to computability judgments by internalizing $A\ \mathit{comp}\ @\ \omega$ with a modality $\bigcirc$ so that $\Gamma \vdash_{\mathsf{s}} \bigcirc A\ \mathit{true}\ @\ \omega$ expresses $\Gamma \vdash_{\mathsf{s}} A\ \mathit{comp}\ @\ \omega$. The introduction and elimination rules use a *computation term* $\mathsf{cmp}\ E$ and a *bind expression* $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$:

$$\frac{\Gamma \vdash_{\mathsf{s}} E \div A\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{cmp}\ E : \bigcirc A\ @\ \omega}\ \bigcirc\mathsf{I} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : \bigcirc A\ @\ \omega \quad \Gamma, x : A \vdash_{\mathsf{s}} E \div B\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E \div B\ @\ \omega}\ \bigcirc\mathsf{E}$$

We use a reduction relation $\Rightarrow_{\beta\mathsf{exp}}$ in both the expression reduction rule for $\bigcirc$ and its corresponding proof reduction:

$$\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F \ \Rightarrow_{\beta\mathsf{exp}}\ \langle E/x\rangle F \qquad (\beta_{\bigcirc})$$

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{s}} E \div A @ \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{cmp}\ E : \bigcirc A @ \omega}\ \bigcirc\mathsf{I} \quad \Gamma, x : A \vdash_{\mathsf{s}} F \div B @ \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F \div B @ \omega}\ \bigcirc\mathsf{E} \quad \Rightarrow_{\beta\mathsf{exp}} \quad \Gamma \vdash_{\mathsf{s}} \langle E/x \rangle F \div B @ \omega$$

$\mathsf{cmp}\ E$ denotes the computation of E, but does not actually compute E; hence we say that $\mathsf{cmp}\ E$ *encapsulates* the computation of E. $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$ enables us to sequence two computations (if M evaluates to a computation term).

Note that the typing rule $\bigcirc\mathsf{E}$ does not accurately reflect the operational behavior of $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$. Specifically, while the rule $\bigcirc\mathsf{E}$ typechecks E at the same world ω that it typechecks M, the computation of E may take place at a different world ω' (where $\omega \leq \omega'$) because of an expression computation preceding the computation of E. Nevertheless it is a sound typing rule because the monotonicity of the accessibility relation $\leq$ allows the type system to reason as if a world effect did not cause a transition to another world, as clarified in the characterization of computability judgments.

Computation terms and bind expressions may be thought of as monadic constructs, since the modality $\bigcirc$ forms a monad. In Haskell syntax, the monad could be written as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{instance\ Monad}\ \bigcirc\ \mathsf{where}} \\ \quad \mathsf{return}\ M & = & \mathsf{cmp}\ \ M \\ \quad M >\!>\!= N & = & \mathsf{cmp}\ \ \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in} \\ & & \qquad\ \ \mathsf{letcmp}\ y \triangleleft N\ x\ \mathsf{in} \\ & & \qquad\ \ y \end{array}$$

The above definition satisfies the monadic laws [77], modulo the expression reduction rule $\beta_{\bigcirc}$ and a term expansion rule $\gamma_{\bigcirc}$ for the modality $\bigcirc$:

$$M \ \Rightarrow_{\eta\mathsf{exp}}\ \mathsf{cmp}\ \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ x \qquad\qquad (\gamma_{\bigcirc})$$

However, once we introduce a fixed point construct for terms, the rule $\gamma_{\bigcirc}$ becomes invalid. For example, if M is a fixed point construct whose reduction never terminates, its expansion into $\mathsf{cmp}\ \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ x$ is not justified because the reduction of the expanded term immediately terminates. Hence the modality $\bigcirc$ ceases to form a monad, and we do not call $\lambda_{\bigcirc}$ a monadic language.

2.3.3 Substitutions

Now that all term and expression constructs have been introduced, we define term and expression substitutions. We first consider term substitutions, which are essentially textual substitutions.

Term substitution

Term substitutions $[M/x]N$ and $[M/x]E$ are straightforward to define as they correspond to substituting a proof of A *true* @ ω for a hypothesis in a hypothetical proof. To formally define term substitutions, we need a mapping $FV(\cdot)$ for obtaining the set of *free variables* in a given term or expression; a free variable is one that is not bound in lambda abstractions and bind expressions:

$$\begin{array}{lll} FV(x) & = & \{x\} \\ FV(\lambda x\!:\!A.\, M) & = & FV(M) - \{x\} \\ FV(M_1\ M_2) & = & FV(M_1) \cup FV(M_2) \\ FV(\mathsf{cmp}\ E) & = & FV(E) \\ FV(\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E) & = & FV(M) \cup (FV(E) - \{x\}) \end{array}$$

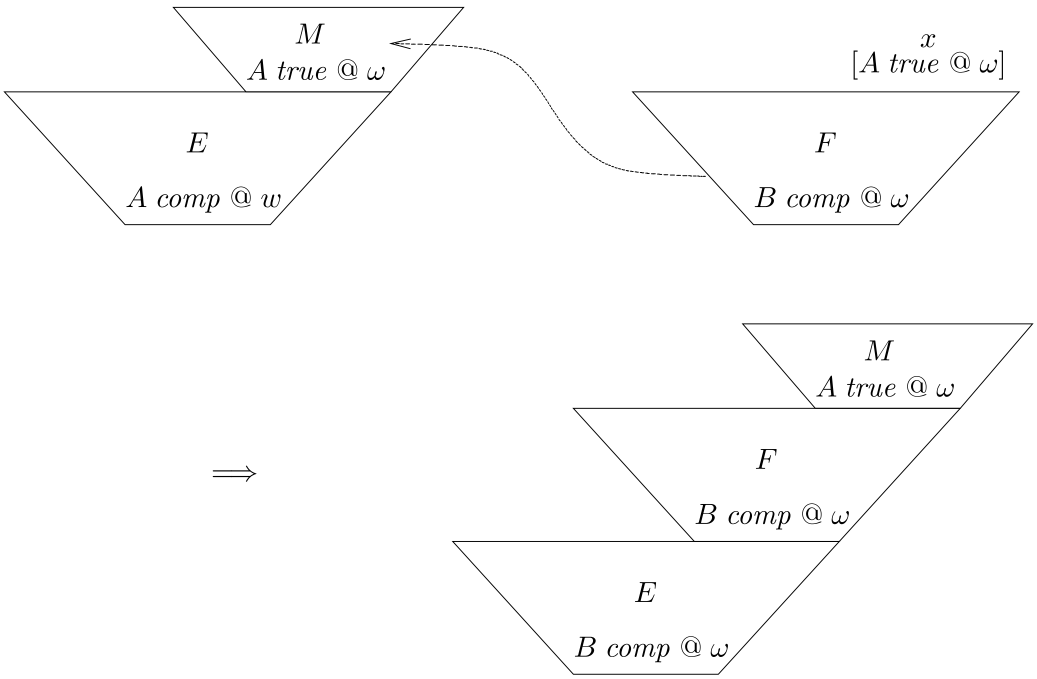


Figure 2.4: A schematic view of $\langle E/x \rangle F$.

In the definition of $[M/x]N$ and $[M/x]E$, we implicitly rename bound variables in N and E as necessary to avoid the capture of free variables in M:[4]

$$\begin{array}{rcll}
[M/x]y & = & M & x = y \\
 & = & y & \textit{otherwise} \\
[M/x]\lambda y\!:\!A.\,N & = & \lambda y\!:\!A.\,[M/x]N & x \neq y, y \notin FV(M) \\
[M/x](N_1\ N_2) & = & [M/x]N_1\ [M/x]N_2 & \\
[M/x]\mathsf{cmp}\ E & = & \mathsf{cmp}\ [M/x]E & \\
[M/x]\mathsf{letcmp}\ y \triangleleft N\ \mathsf{in}\ E & = & \mathsf{letcmp}\ y \triangleleft [M/x]N\ \mathsf{in}\ [M/x]E & x \neq y, y \notin FV(M)
\end{array}$$

The above definition of term substitution conforms to the substitution principle for terms:

Proposition 2.3 (Substitution principle for terms).
If $\Gamma \vdash_{\mathsf{s}} M : A @ \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} N : B @ \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [M/x]N : B @ \omega$.
If $\Gamma \vdash_{\mathsf{s}} M : A @ \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} E \div B @ \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [M/x]E \div B @ \omega$.

Proof. By simultaneous induction on the structure of N and E. □

Proposition 2.3 implies that term reductions by $\Rightarrow_{\beta\mathsf{term}}$ are indeed type-preserving:

Corollary 2.4 (Type preservation of $\Rightarrow_{\beta\mathsf{term}}$).
If $\Gamma \vdash_{\mathsf{s}} (\lambda x\!:\!A.\,N)\ M : B @ \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [M/x]N : B @ \omega$.

[4]Hence a term substitution does not need to be defined in all cases.

Expression substitution

Given $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ and $\Gamma, x : A \vdash_{\mathsf{s}} F \div B \mathbin{@} \omega$, an expression substitution combines the two typing judgments by finding an expression $\langle E/x \rangle F$ such that $\Gamma \vdash_{\mathsf{s}} \langle E/x \rangle F \div B \mathbin{@} \omega$. It corresponds to substituting a hypothetical proof using A *true* $@\ \omega$ as a hypothesis into a proof of A *comp* $@\ \omega$.

Figure 2.4 shows a schematic view of an expression substitution $\langle E/x \rangle F$. Expression E contains a term M of type A which ultimately determines its type. For example, $E = \mathsf{letcmp}\ x \triangleleft N\ \mathsf{in}\ M$ has the same type as M, and if M is replaced by another expression E' of type A', the resultant expression also has type A'. Operationally the computation of E finishes by evaluating M. Expression F contains variable x which corresponds to a hypothesis A *true* $@\ \omega$ in a hypothetical proof of B *comp* $@\ \omega$. $\langle E/x \rangle F$ first substitutes M for x in F, which results in a new expression $[M/x]F$ of type B; then it replaces M in E by $[M/x]F$. In this way, $\langle E/x \rangle F$ substitutes F into E, rather than E into F. Note that although $\langle E/x \rangle F$ transforms the structure of E, it has the same type as F because its type is ultimately determined by whatever expression replaces M.

Thus $\langle E/x \rangle F$ analyzes the structure of E, instead of F, to find a term that ultimately determines the type of E:

$$\begin{array}{rcl} \langle M/x \rangle F & = & [M/x]F \\ \langle \mathsf{letcmp}\ y \triangleleft M\ \mathsf{in}\ E'/x \rangle F & = & \mathsf{letcmp}\ y \triangleleft M\ \mathsf{in}\ \langle E'/x \rangle F \end{array}$$

The above definition of expression substitution conforms to the substitution principle for expressions:

Proposition 2.5 (Substitution principle for expressions).
If $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ *and* $\Gamma, x : A \vdash_{\mathsf{s}} F \div B \mathbin{@} \omega$*, then* $\Gamma \vdash_{\mathsf{s}} \langle E/x \rangle F \div B \mathbin{@} \omega$.

Proof. By induction on the structure of E (not F). □

Proposition 2.5 implies that expression reductions by $\Rightarrow_{\beta\mathsf{exp}}$ are indeed type-preserving:

Corollary 2.6 (Type preservation of $\Rightarrow_{\beta\mathsf{exp}}$).
If $\Gamma \vdash_{\mathsf{s}} \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F \div B \mathbin{@} \omega$*, then* $\Gamma \vdash_{\mathsf{s}} \langle E/x \rangle F \div B \mathbin{@} \omega$.

2.3.4 World terms and instructions

The operational semantics of $\lambda_{\bigcirc}$ provides rules for term evaluations $M \rightharpoonup V$ and expression computations $E \mathbin{@} \omega \rightharpoondown V \mathbin{@} \omega'$. For term evaluations, we introduce a term reduction $M \mapsto_{\mathsf{t}} N$ such that $M \mapsto_{\mathsf{t}}^{*} V$ is identified with $M \rightharpoonup V$, where $\mapsto_{\mathsf{t}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{t}}$; for expression computations, we introduce an expression reduction $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$ such that $E \mathbin{@} \omega \mapsto_{\mathsf{e}}^{*} V \mathbin{@} \omega'$ is identified with $E \mathbin{@} \omega \rightharpoondown V \mathbin{@} \omega'$, where $\mapsto_{\mathsf{e}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{e}}$:

$$M \mapsto_{\mathsf{t}}^{*} V \quad \textit{iff} \quad M \rightharpoonup V \qquad\qquad E \mathbin{@} \omega \mapsto_{\mathsf{e}}^{*} V \mathbin{@} \omega' \quad \textit{iff} \quad E \mathbin{@} \omega \rightharpoondown V \mathbin{@} \omega'$$

At this point, there is no language construct for producing world effects and no typing rules and reduction rules actually require worlds. That is, all language constructs introduced so far are purely logical in that their definition is explained either by properties of judgments (*e.g.*, variables, inclusion of terms into expressions) or by introduction and elimination rules (*e.g.*, lambda abstractions, lambda applications). In fact, if we erase $@\ \omega$ from typing judgments, $\lambda_{\bigcirc}$ reverts to the monadic language of Pfenning and Davies [60]. Thus we introduce language constructs for interacting with worlds before presenting the operational semantics.

We use *instructions* I as expressions for producing world effects. As an interface to worlds, they are provided by the programming environment. For example, an instruction $\mathsf{new}\ M$ for allocating new references produces a world effect by causing a change to the store, and returns a reference. An instruction may have arguments, and term substitution on instructions with arguments is defined in a structural way; hence Proposition 2.3 continues to hold.

We refer to those objects originating from worlds, such as references, as *world terms* W. Since they cannot be decomposed into ordinary terms, world terms are assumed to be atomic values (containing no subterms) and are given special *world term types* $\mathcal{W}$. For example, reference type $\mathsf{ref}\ A$ is a world term type for references. Note that while world terms may not contain ordinary terms, world term types may contain ordinary types (*e.g.*, $\mathsf{ref}\ A$).

The new abstract syntax for $\lambda_{\bigcirc}$ is as follows:

$$
\begin{array}{lrcl}
\text{type} & A & ::= & \cdots \mid \mathcal{W} \\
\text{world term type} & \mathcal{W} & & \\
\text{term} & M & ::= & \cdots \mid W \\
\text{world term} & W & & \\
\text{expression} & E & ::= & \cdots \mid I \\
\text{instruction} & I & & \\
\text{value} & V & ::= & \cdots \mid W
\end{array}
$$

The type of a world term may depend on the world where it resides. For example, a reference is a pointer to a heap cell and its type depends on the store for which it is valid. Therefore typing rules for world terms may have to analyze worlds. Since world terms are atomic values, typing judgments for world terms do not require typing contexts. In contrast, typing judgments for instructions require typing contexts because instructions may include terms as arguments:

$$W : \mathcal{W} \ @\ \omega \qquad\qquad \Gamma \vdash_{\mathsf{s}} I \div A \ @\ \omega$$

Note that an instruction does not necessarily have a world term type. For example, an instruction for dereferencing references can have any type because heap cells can contain values of any type.

If an instruction I whose arguments are all values typechecks at a world ω under an empty typing context, we regard it as reducible at ω; moreover we require that an instruction reduction $I \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega'$ be type-preserving so that V has the same type as I:

Type-preservation/progress requirement on instructions

If $\cdot \vdash_{\mathsf{s}} I \div A \ @\ \omega$ *and arguments to* I *are all values, then there exists a world* ω' *satisfying* $I \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega'$ *and* $\cdot \vdash_{\mathsf{s}} V : A \ @\ \omega'$.

We allow $\omega = \omega'$, which means that a world effect does not always causes a change to a world (*e.g.*, reading the contents of a store is still a world effect).

As $I \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega'$ means that instruction I computes to value V causing a transition of world from ω to ω', it implies $\omega \leq \omega'$. Now the accessibility relation $\leq$ is fully specified by instruction reductions under the assumption that it is reflexive and transitive. Note that without additional requirements on instructions, there is no guarantee that the monotonicity of $\leq$ is maintained. For example, an instruction for deallocating an existing reference l violates monotonicity of truth if l no longer typechecks after it is deallocated, and violates persistence of computation if its corresponding heap cell is discarded. In order to maintain the monotonicity of $\leq$, we further require that all instruction reductions be designed in such a way that types of world terms and instructions are unaffected by $\leq$:

Monotonicity requirement on instructions

1) If $\omega \leq \omega'$, *then* $W : \mathcal{W} \ @\ \omega$ *implies* $W : \mathcal{W} \ @\ \omega'$.

2) If $\omega \leq \omega'$, *then* $\Gamma \vdash_{\mathsf{s}} I \div A \ @\ \omega$ *implies* $\Gamma \vdash_{\mathsf{s}} I \div A \ @\ \omega'$, *where for each argument* M *to* I, *we assume that* $\Gamma \vdash_{\mathsf{s}} M : B \ @\ \omega$ *implies* $\Gamma \vdash_{\mathsf{s}} M : B \ @\ \omega'$.

$$\frac{M \mapsto_{\mathsf{t}} M'}{M\ N \mapsto_{\mathsf{t}} M'\ N}\ T_{\beta_L} \qquad \frac{}{(\lambda x\!:\!A.\,M)\ N \mapsto_{\mathsf{t}} [N/x]M}\ T_{\beta} \qquad \frac{M \mapsto_{\mathsf{t}} N}{M \mathbin{@} \omega \mapsto_{\mathsf{e}} N \mathbin{@} \omega}\ E_{Term}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft N\ \mathsf{in}\ F \mathbin{@} \omega}\ E_{Bind}$$

$$\frac{E \neq I}{\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} \langle E/x\rangle F \mathbin{@} \omega}\ E_{Bind\beta}$$

$$\frac{I \mathbin{@} \omega \mapsto_{\mathsf{e}} V \mathbin{@} \omega'}{\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ I\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ V\ \mathsf{in}\ F \mathbin{@} \omega'}\ E_{BindI}$$

Figure 2.5: Operational semantics of $\lambda_{\bigcirc}$ which uses expression substitutions for expression computations.

The first clause corresponds to monotonicity of truth, and the second clause to persistence of computation. Under the monotonicity requirement, instruction reductions never affect types of existing terms and expressions:

Proposition 2.7 (Monotonicity of $\leq$).
If $\omega \leq \omega'$, then
$\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ *implies* $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega'$, *and*
$\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ *implies* $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega'$.

Proof. By simultaneous induction on the structure of M and E. □

Unlike other expression constructs, instructions are not explained logically and no expression substitution can be defined on them. Intuitively $\langle I/x\rangle E$ cannot be reduced into another expression because I itself does not reveal a term that is evaluated at the end of its computation. Such a term (which is indeed a value) becomes known only after an instruction reduction $I \mathbin{@} \omega \mapsto_{\mathsf{e}} V \mathbin{@} \omega'$. We should therefore never attempt to directly reduce $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ I\ \mathsf{in}\ E$ into $\langle I/x\rangle E$. For the sake of convenience and uniform notation, however, we abuse the notation $\langle I/x\rangle E$ with the following definition, which effectively prevents $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ I\ \mathsf{in}\ E$ from being reduced by $\Rightarrow_{\beta\mathsf{exp}}$:

$$\langle I/x\rangle E \quad = \quad \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ I\ \mathsf{in}\ E$$

This definition of $\langle I/x\rangle E$ allows $\Rightarrow_{\beta\mathsf{exp}}$ to be applied to any part of a given expression; Proposition 2.5 also continues to hold.

2.3.5 Operational semantics

A term reduction by $\Rightarrow_{\beta\mathsf{term}}$ and an expression reduction by $\Rightarrow_{\beta\mathsf{exp}}$ are both proof reductions and may be applied to any part of a given term or expression without affecting its type. An operational semantics of $\lambda_{\bigcirc}$ defines the term reduction relation $\mapsto_{\mathsf{t}}$ and the expression reduction relation $\mapsto_{\mathsf{e}}$ by specifying a strategy for arranging reductions by $\Rightarrow_{\beta\mathsf{term}}$ and $\Rightarrow_{\beta\mathsf{exp}}$. Below we consider two different styles of operational semantics (both of which use the same syntax for reduction relations). For each instruction I, we assume an instruction reduction $I \mathbin{@} \omega \mapsto_{\mathsf{e}} V \mathbin{@} \omega'$, which causes a transition of world from ω to ω'; if I has arguments, we first reduce them into values by applying $\mapsto_{\mathsf{t}}$ repeatedly.

Figure 2.5 shows an operational semantics of $\lambda_{\bigcirc}$ which uses expression substitutions $\langle E/x\rangle F$ for expression computations; for term evaluations, we can choose any reduction strategy (Figure 2.5 uses a call-by-name discipline). The rule T_{β} is a shorthand for applying $\Rightarrow_{\beta\mathsf{term}}$ to $(\lambda x\!:\!A.\,M)\ N$. The rules E_{Term}

$$\frac{M \mapsto_{\mathsf{t}} M'}{M\ N \mapsto_{\mathsf{t}} M'\ N}\ T_{\beta_L} \qquad \frac{N \mapsto_{\mathsf{t}} N'}{(\lambda x\!:\!A.\,M)\ N \mapsto_{\mathsf{t}} (\lambda x\!:\!A.\,M)\ N'}\ T_{\beta_R}$$

$$\frac{}{(\lambda x\!:\!A.\,M)\ V \mapsto_{\mathsf{t}} [V/x]M}\ T_{\beta_V} \qquad \frac{M \mapsto_{\mathsf{t}} N}{M \mathbin{@} \omega \mapsto_{\mathsf{e}} N \mathbin{@} \omega}\ E_{Term}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft N\ \mathsf{in}\ F \mathbin{@} \omega}\ E_{Bind}$$

$$\frac{E \mathbin{@} \omega \mapsto_{\mathsf{e}} E' \mathbin{@} \omega'}{\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E'\ \mathsf{in}\ F \mathbin{@} \omega'}\ E_{BindR}$$

$$\frac{}{\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ V\ \mathsf{in}\ F \mathbin{@} \omega \mapsto_{\mathsf{e}} [V/x]F \mathbin{@} \omega}\ E_{BindV}$$

Figure 2.6: Operational semantics of $\lambda_{\bigcirc}$ in the direct style.

and E_{Bind} use a term reduction $M \mapsto_{\mathsf{t}} N$ to reduce a term into a value. The rule $E_{Bind\beta}$ is a shorthand for applying $\Rightarrow_{\beta\mathsf{exp}}$ to $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F$; in the case of $E = M$, it reduces $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ M\ \mathsf{in}\ F$ into $\langle M/x \rangle F = [M/x]F$ without further reducing M. The rule E_{BindI} perform an instruction reduction $I \mathbin{@} \omega \mapsto_{\mathsf{e}} V \mathbin{@} \omega'$.

Figure 2.6 shows an alternative style of operational semantics, called the direct style, which requires only term substitutions $[V/x]E$ for expression computations; for term evaluations, we can choose any reduction strategy (Figure 2.6 uses a call-by-value discipline). The rules E_{Term} and E_{Bind} are the same as in Figure 2.5. Given $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ E\ \mathsf{in}\ F$, we apply the rule E_{BindR} repeatedly until E is reduced into a value V; then the rule E_{BindV} reduces $\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ V\ \mathsf{in}\ F$ into $\langle V/x \rangle F = [V/x]F$. Thus a variable is always replaced by a value (during both term evaluations and expression computations).

The direct style is more extensible than the first style because it does not use expression substitutions. That is, the introduction of a new expression construct requires only new reduction rules. In comparison, the first style hinges on expression substitutions, and requires not only new reduction rules but also an augmented definition of expression substitution for each new expression construct. If expression substitution cannot be defined on a new expression construct, we may have to further specialize existing reduction rules. For example, the rules $E_{Bind\beta}$ and E_{BindI} can be thought of as derived from a common reduction rule when instructions are introduced.

The type safety of $\lambda_{\bigcirc}$ consists of two properties: type preservation and progress. The proof of type preservation uses Corollaries 2.4 and 2.6, the type-preservation/progress requirement on instructions, and Proposition 2.7. The proof of progress requires a canonical forms lemma. In either style of the operational semantics, all proofs proceed in the same way.

Theorem 2.8 (Type preservation).

If $M \mapsto_{\mathsf{t}} N$ and $\cdot \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$, then $\cdot \vdash_{\mathsf{s}} N : A \mathbin{@} \omega$.
If $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$ and $\cdot \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$, then $\cdot \vdash_{\mathsf{s}} F \div A \mathbin{@} \omega'$.

Proof. By induction on the structure of M and E. □

Lemma 2.9 (Canonical forms).

If V is a value of type $A \supset B$, then V is a lambda abstraction $\lambda x\!:\!A.\,M$.
If V is a value of type $\bigcirc A$, then V is a computation term $\mathsf{cmp}\ E$.

Proof. By inspection of the typing rules. □

Theorem 2.10 (Progress).
If $\cdot \vdash_{\mathsf{s}} M : A \text{ @ } \omega$, *then either* M *is a value or there exists* N *such that* $M \mapsto_{\mathsf{t}} N$.
If $\cdot \vdash_{\mathsf{s}} E \div A \text{ @ } \omega$, *then either* E *is a value or there exist* F *and* ω' *such that* $E \text{ @ } \omega \mapsto_{\mathsf{e}} F \text{ @ } \omega'$.

Proof. By induction on the structure of M and E. □

Since expressions may produce world effects, they cannot be converted into terms. In contrast, terms can always be lifted to expressions by the typing rule Term. Therefore we define a program as a closed expression E that typechecks at a certain initial world $\omega_{initial}$, *i.e.*, $\cdot \vdash_{\mathsf{s}} E \div A \text{ @ } \omega_{initial}$. We choose $\omega_{initial}$ according to the world structure being employed. To run a program E, we compute it at $\omega_{initial}$.

2.4 Examples of world effects

In order to implement a specific notion of world effect in $\lambda_{\bigcirc}$, we specify a world structure and provide instructions to interact with worlds. In this section, we discuss three specific notions of world effect.

2.4.1 Probabilistic computations

In order to facilitate the coding of sampling techniques developed in simulation theory, we model a probabilistic computation as a computation that returns a value after consuming real numbers drawn independently from $U(0.0, 1.0]$, rather than a single such real number. A real number r is a world term of type real. A world, the source of probabilistic choices, is represented as an infinite sequence of real numbers drawn independently from $U(0.0, 1.0]$. We use an instruction $\mathcal{S}$ for consuming the first real number of a given world.

$$
\begin{array}{llll}
\text{world term type} & \mathcal{W} & ::= & \mathsf{real} \\
\text{world term} & W & ::= & r \\
\text{instruction} & I & ::= & \mathcal{S} \\
\text{world} & \omega & ::= & r_1 r_2 \cdots r_i \cdots \quad \textit{where } r_i \in (0.0, 1.0]
\end{array}
$$

$$
\frac{}{r : \mathsf{real} \text{ @ } \omega}\ \mathsf{Real} \qquad \frac{}{\Gamma \vdash_{\mathsf{s}} \mathcal{S} \div \mathsf{real} \text{ @ } \omega}\ \mathsf{Sampling}
$$

$$
\frac{}{\mathcal{S} \text{ @ } r_1 r_2 r_3 \cdots \mapsto_{\mathsf{e}} r_1 \text{ @ } r_2 r_3 \cdots}\ \mathit{Sampling}
$$

It is easy to show that instruction $\mathcal{S}$ satisfies the type-preservation/progress requirement. Since a world does not affect types of world terms and instructions, the monotonicity of $\leq$ also holds trivially. We can use any world as an initial world. As we will see in Chapter 3, $\lambda_{\bigcirc}$ with the above constructs for probabilistic computations serves as the core of PTP.

2.4.2 Sequential input/output

We model sequential input/output with a computation that consumes an infinite input character stream is and outputs to a finite output character stream os, where a character is a world term of type char. We use two instructions: read_c for reading a character from the input stream and write_c M for writing a character to the output stream.

$$
\begin{array}{llll}
\text{world term type} & \mathcal{W} & ::= & \mathsf{char} \\
\text{world term} & W & ::= & c \\
\text{instruction} & I & ::= & \mathsf{read\_c} \mid \mathsf{write\_c}\ M \\
\text{world} & \omega & ::= & (is, os) \\
 & is & ::= & c_1 c_2 c_3 \cdots \\
 & os & ::= & \mathsf{nil} \mid c :: os
\end{array}
$$

$$\frac{}{c : \mathsf{char} \mathbin{@} \omega}\ \mathsf{Char} \qquad \frac{}{\Gamma \vdash_{\mathsf{s}} \mathsf{read\_c} \div \mathsf{char} \mathbin{@} \omega}\ \mathsf{Read\_c}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M : \mathsf{char} \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{write\_c}\ M \div \mathsf{char} \mathbin{@} \omega}\ \mathsf{Write\_c}$$

$$\frac{}{\mathsf{read\_c} \mathbin{@} (c_1 c_2 c_3 \cdots, os) \mapsto_{\mathsf{e}} c_1 \mathbin{@} (c_2 c_3 \cdots, os)}\ \mathit{Read\_c}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{write\_c}\ M \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{write\_c}\ N \mathbin{@} \omega}\ \mathit{Write\_c}$$

$$\frac{}{\mathsf{write\_c}\ c \mathbin{@} (is, os) \mapsto_{\mathsf{e}} c \mathbin{@} (is, c :: os)}\ \mathit{Write\_c}'$$

It is easy to show that both instructions satisfy the type-preservation/progress requirement. As in probabilistic computations, a world does not affect types of world terms and instructions, and the monotonicity of $\leq$ holds trivially. We use an empty output character stream nil in an initial world.

2.4.3 Mutable references

Probabilistic computations and sequential input/output are easy to model because worlds do not affect types of world terms and instructions. Mutable references, however, require world terms whose type depends on worlds, namely references. Consequently worlds should be designed in such a way that they provide enough information on a given reference to correctly determine its type.

We use $\mathsf{ref}\ A$ as world term types for references. A world is represented as a collection of pairs $[l \mapsto V : A]$ of a reference l and a closed value V annotated with its type A. It may be thought of as a well-typed store: if $[l \mapsto V : A] \in \omega$, then V has type A at world ω (*i.e.*, $\cdot \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$) and references in it are all distinct. We use three instructions: $\mathsf{new}\ M : A$ for initializing a fresh reference, $\mathsf{read}\ M$ for reading the contents of a world, and $\mathsf{write}\ M\ M$ for updating a world. Reading the contents of a world is a world effect, even though it does not cause a change to the world.

world term type	$\mathcal{W}$	$::=$	$\mathsf{ref}\ A$
world term	W	$::=$	l
instruction	I	$::=$	$\mathsf{new}\ M : A \mid \mathsf{read}\ M \mid \mathsf{write}\ M\ M$
world	ω	$::=$	$\cdot \mid \omega, [l \mapsto V : A]$

Figure 2.7 shows new typing rules and reduction rules:

To prove the type-preservation/progress requirement on instructions, we first show that well-typed instructions never generate corrupt worlds (Corollaries 2.12 and 2.14). In Lemma 2.11, we do not postulate that $\omega, [l \mapsto V : A]$ is a world (*i.e.*, it possesses the structure of a store, but may not be well-typed).

Lemma 2.11. *If ω is a world and $\cdot \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$, then*
$\Gamma \vdash_{\mathsf{s}} M : B \mathbin{@} \omega$ implies $\Gamma \vdash_{\mathsf{s}} M : B \mathbin{@} \omega, [l \mapsto V : A]$, and
$\Gamma \vdash_{\mathsf{s}} E \div B \mathbin{@} \omega$ implies $\Gamma \vdash_{\mathsf{s}} E \div B \mathbin{@} \omega, [l \mapsto V : A]$, where l is a fresh reference.

Proof. By simultaneous induction on the structure of M and E. An interesting case is when $M = l' \neq l$.

If $M = l'$, then $\Gamma \vdash_{\mathsf{s}} M : B \mathbin{@} \omega$ implies $B = \mathsf{ref}\ B'$ and $[l' \mapsto V' : B'] \in \omega$ by the rule Ref. Since $[l' \mapsto V' : B'] \in \omega, [l \mapsto V : A]$, we have $\Gamma \vdash_{\mathsf{s}} M : B \mathbin{@} \omega, [l \mapsto V : A]$. ☐

Corollary 2.12. *If $\cdot \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$ where ω is a world, then $\omega, [l \mapsto V : A]$ is also a world for any fresh reference l.*

Proof. For each $[l' \mapsto V' : A'] \in \omega$, we have $\cdot \vdash_{\mathsf{s}} V' : A' \mathbin{@} \omega$ because ω is a world. By Lemma 2.11, we have $\cdot \vdash_{\mathsf{s}} V' : A' \mathbin{@} \omega, [l \mapsto V : A]$. From $\cdot \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$ and Lemma 2.11, $\cdot \vdash_{\mathsf{s}} V : A \mathbin{@} \omega, [l \mapsto V : A]$ also follows. ☐

$$\frac{[l \mapsto V : A] \in \omega}{l : \mathsf{ref}\ A \ @\ \omega}\ \mathsf{Ref} \qquad \frac{\Gamma \vDash_{\mathsf{s}} M : A \ @\ \omega}{\Gamma \vDash_{\mathsf{s}} \mathsf{new}\ M : A \div \mathsf{ref}\ A \ @\ \omega}\ \mathsf{New}$$

$$\frac{\Gamma \vDash_{\mathsf{s}} M : \mathsf{ref}\ A \ @\ \omega}{\Gamma \vDash_{\mathsf{s}} \mathsf{read}\ M \div A \ @\ \omega}\ \mathsf{Read} \qquad \frac{\Gamma \vDash_{\mathsf{s}} M : \mathsf{ref}\ A \ @\ \omega \quad \Gamma \vDash_{\mathsf{s}} N : A \ @\ \omega}{\Gamma \vDash_{\mathsf{s}} \mathsf{write}\ M\ N \div A \ @\ \omega}\ \mathsf{Write}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{new}\ M : A \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{new}\ N : A \ @\ \omega}\ \mathit{New}$$

$$\frac{\mathit{fresh}\ l\ \mathit{such\ that}\ [l \mapsto V' : A'] \notin \omega}{\mathsf{new}\ V : A \ @\ \omega \mapsto_{\mathsf{e}} l \ @\ \omega, [l \mapsto V : A]}\ \mathit{New}'$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{read}\ M \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{read}\ N \ @\ \omega}\ \mathit{Read} \qquad \frac{[l \mapsto V : A] \in \omega}{\mathsf{read}\ l \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega}\ \mathit{Read}'$$

$$\frac{M \mapsto_{\mathsf{t}} M'}{\mathsf{write}\ M\ N \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{write}\ M'\ N \ @\ \omega}\ \mathit{Write}$$

$$\frac{N \mapsto_{\mathsf{t}} N'}{\mathsf{write}\ l\ N \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{write}\ l\ N' \ @\ \omega}\ \mathit{Write}'$$

$$\frac{[l \mapsto V' : A] \in \omega}{\mathsf{write}\ l\ V \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega - [l \mapsto V' : A], [l \mapsto V : A]}\ \mathit{Write}''$$

Figure 2.7: Typing rules and reduction rules for mutable references.

In Lemma 2.13, we do not postulate that $\omega - [l \mapsto V' : A], [l \mapsto V : A]$ is a world.

Lemma 2.13.
If $\cdot \vDash_{\mathsf{s}} V : A \ @\ \omega$ *and* $[l \mapsto V' : A] \in \omega$ *where* ω *is a world, then*
$\Gamma \vDash_{\mathsf{s}} M : B \ @\ \omega$ *implies* $\Gamma \vDash_{\mathsf{s}} M : B \ @\ \omega - [l \mapsto V' : A], [l \mapsto V : A]$ *and*
$\Gamma \vDash_{\mathsf{s}} E \div B \ @\ \omega$ *implies* $\Gamma \vDash_{\mathsf{s}} E \div B \ @\ \omega - [l \mapsto V' : A], [l \mapsto V : A]$.

Proof. By simultaneous induction on the structure of M and E. An interesting case is when $M = l$. □

Corollary 2.14.
If $\cdot \vDash_{\mathsf{s}} V : A \ @\ \omega$ *and* $[l \mapsto V' : A] \in \omega$ *where* ω *is a world, then*
$\omega - [l \mapsto V' : A], [l \mapsto V : A]$ *is also a world.*

Proof. Similarly to the proof of Corollary 2.12. □

Proposition 2.15 (Type-preservation/progress requirement on instructions). *If* $\cdot \vDash_{\mathsf{s}} I \div A \ @\ \omega$ *and arguments to* I *are all values, then there exists a world* ω' *satisfying* $I \ @\ \omega \mapsto_{\mathsf{e}} V \ @\ \omega'$ *and* $\cdot \vDash_{\mathsf{s}} V : A \ @\ \omega'$.

Proof. By case analysis of I. We use Corollaries 2.12 and 2.14. □

For the monotonicity requirement on instructions, we directly prove Proposition 2.7 exploiting Lemmas 2.11 and 2.13.

Proof of Proposition 2.7. Since the accessibility relation $\leq$ is specified by instruction reductions, $\omega \leq \omega'$ implies that

$$\omega = \omega_1 \leq \cdots \leq \omega_i \leq \cdots \leq \omega_n = \omega',$$

where ω_{i+1} is equal to either $\omega_i, [l \mapsto V : A]$ or $\omega_i - [l \mapsto V' : A], [l \mapsto V : A]$ for $1 \leq i < n$. We proceed by induction on n. □

In order to maintain the monotonicity of $\leq$, all references in a world must be persistent, since once a reference is deallocated, its type can no longer be determined. This means that an explicit instruction for deallocating references (*e.g.*, $\mathsf{delete}\ M$) is not allowed in $\lambda_{\bigcirc}$. In the present framework of $\lambda_{\bigcirc}$, even garbage collections are not allowed because they destroy the monotonicity of $\leq$: a garbage collection transition from ω to ω' must ensure that $l : \mathsf{ref}\ A\ @\ \omega$ implies $l : \mathsf{ref}\ A\ @\ \omega'$ for every possible reference l, including those references not found in a given program, which are precisely what it deallocates. (In practice, garbage collections do not interfere with evaluations and computations, and are safe to implement.) We use an empty store as an initial world.

2.4.4 Supporting multiple notions of world effect

Since a world structure realizes a specific notion of world effect and instructions provide an interface to worlds, we can support multiple notions of world effect by combining individual world structures and letting each instruction interact with its relevant part of worlds. For example, we can use all the above instructions if a world consists of three sub-worlds: an infinite sequence of real numbers, input/output streams, and a well-typed store. This is how $\lambda_{\bigcirc}$ combines world effects at the language design level.

We may think of $\lambda_{\bigcirc}$ as providing a built-in implementation of a state monad whose states are worlds. Then the ease of combining world effects in $\lambda_{\bigcirc}$ reflects the fact that state monads combine well with each other (by combining individual states).

2.5 Fixed point constructs

In this section, we investigate an extension of $\lambda_{\bigcirc}$ with fixed point constructs. We first consider those based upon the unfolding semantics, in which a fixed point construct reduces by unrolling itself. Next we consider those based upon the backpatching semantics, as used in Scheme [3]. For expressions, we assume the operational semantics in the direct style in Figure 2.6.

For a uniform treatment of types, we choose to allow fixed point constructs for all types. An alternative approach would be to confine fixed point constructs only to lambda abstractions (as in ML), but it would be inadequate for our purpose because recursive computations require fixed point constructs for computation terms (of type $\bigcirc A$) anyway.

2.5.1 Unfolding semantics

We use $\mathsf{fix}\ x\!:\!A.\ M$ as a *term fixed point construct* for recursive evaluations. Its typing rule and reduction rule are as usual:

$$\text{term} \quad M \quad ::= \quad \cdots \mid \mathsf{fix}\ x\!:\!A.\ M$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} M : A\ @\ \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{fix}\ x\!:\!A.\ M : A\ @\ \omega}\ \mathsf{Fix} \qquad \frac{}{\mathsf{fix}\ x\!:\!A.\ M \mapsto_{\mathsf{t}} [\mathsf{fix}\ x\!:\!A.\ M/x]M}\ T_{\mathit{Fix}}$$

In the presence of term fixed point constructs, any truth judgment $A\ \mathit{true}$ holds vacuously, since $\mathsf{fix}\ x\!:\!A.\ x$ typechecks for every type A and represents a proof of $A\ \mathit{true}$. Now a term M of type A does not always represent a constructive proof of $A\ \mathit{true}$; rather it may contain nonsensical proofs such as $\mathsf{fix}\ x\!:\!B.\ x$. The definition of a computability judgment $A\ \mathit{comp}$, however, remains the same because it is defined relative to a truth judgment $A\ \mathit{true}$.

In conjunction with computation terms $\mathsf{cmp}\ E$, term fixed point constructs enable us to encode recursive computations: we first build a term fixed point construct M of type $\bigcirc A$ and then convert it into an expression $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ x$, which denotes a recursive computation. Generalizing this idea, we define syntactic sugar for recursive computations. We introduce an *expression variable* $\mathbf{x}$ and an *expression fixed point construct* $\mathsf{efix}\,\mathbf{x}\div A.\,E$; a new form of binding $\mathbf{x} \div A$ for expression variables is used in typing contexts:

$$\begin{array}{llcl} \text{expression} & E & ::= & \cdots \mid \mathbf{x} \mid \mathsf{efix}\,\mathbf{x}\div A.\,E \\ \text{typing context} & \Gamma & ::= & \cdots \mid \Gamma, \mathbf{x} \div A \end{array}$$

New typing rules and reduction rule are as follows:

$$\frac{}{\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} \mathbf{x} \div A \mathbin{@} \omega}\ \mathsf{Evar} \qquad \frac{\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega}{\Gamma \vdash_{\mathsf{s}} \mathsf{efix}\,\mathbf{x}\div A.\,E \div A \mathbin{@} \omega}\ \mathsf{Efix}$$

$$\frac{}{\mathsf{efix}\,\mathbf{x}\div A.\,E \mathbin{@} \omega \mapsto_{\mathsf{e}} [\mathsf{efix}\,\mathbf{x}\div A.\,E/\mathbf{x}]E \mathbin{@} \omega}\ \mathit{Efix}$$

In the rule Efix, $[\mathsf{efix}\,\mathbf{x}\div A.\,E/\mathbf{x}]E$ denotes a capture-avoiding substitution of $\mathsf{efix}\,\mathbf{x}\div A.\,E$ for expression variable $\mathbf{x}$. Thus $\mathsf{efix}\,\mathbf{x}\div A.\,E$ behaves like term fixed point constructs except that it unrolls itself by substituting an expression for an expression variable, instead of a term for an ordinary variable.

To simulate expression fixed point constructs, we define a function $(\cdot)^\star$ which translates $(\mathsf{efix}\,\mathbf{x}\div A.\,E)^\star$ into:

$$\mathsf{letcmp}\ y_r \triangleleft \mathsf{fix}\ x_p : \bigcirc A.\ \mathsf{cmp}\ [\mathsf{letcmp}\ y_v \triangleleft x_p\ \mathsf{in}\ y_v/\mathbf{x}]E^\star\ \mathsf{in}\ y_r$$

That is, we introduce a variable x_p to encapsulate $\mathsf{efix}\,\mathbf{x}\div A.\,E$ and expand $\mathbf{x}$ to a bind expression $\mathsf{letcmp}\ y_v \triangleleft x_p\ \mathsf{in}\ y_v$. The translation of other terms and expressions is structural; for the sake of simplicity, we do not consider world terms and instructions:

$$\begin{array}{rcl} x^\star & = & x \\ (\lambda x : A.\, M)^\star & = & \lambda x : A.\, M^\star \\ (M_1\ M_2)^\star & = & {M_1}^\star\ {M_2}^\star \\ (\mathsf{cmp}\ E)^\star & = & \mathsf{cmp}\ E^\star \\ (\mathsf{fix}\ x : A.\, M)^\star & = & \mathsf{fix}\ x : A.\, M^\star \\ (\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E)^\star & = & \mathsf{letcmp}\ x \triangleleft M^\star\ \mathsf{in}\ E^\star \\ \mathbf{x}^\star & = & \mathbf{x} \end{array}$$

Proposition 2.17 shows that when translated via the function $(\cdot)^\star$, the typing rules Evar and Efix are sound with respect to the original type system (without the rules Evar and Efix).

Lemma 2.16.
If $\Gamma \vdash_{\mathsf{s}} F \div A \mathbin{@} \omega$ *and* $\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} M : B \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [F/\mathbf{x}]M : B \mathbin{@} \omega$.
If $\Gamma \vdash_{\mathsf{s}} F \div A \mathbin{@} \omega$ *and* $\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} E \div B \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} [F/\mathbf{x}]E \div B \mathbin{@} \omega$.

Proof. By simultaneous induction on the structure of M and E. □

Proposition 2.17.
If $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} M^\star : A \mathbin{@} \omega$.
If $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$, *then* $\Gamma \vdash_{\mathsf{s}} E^\star \div A \mathbin{@} \omega$.

Proof. By simultaneous induction on the structure of the derivation of $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ and $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$. An interesting case is when $E = \mathsf{efix}\,\mathbf{x}\div A.\,F$.
$\mathit{Case}\ E = \mathsf{efix}\,\mathbf{x}\div A.\,F$:
$\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} F \div A \mathbin{@} \omega$ by Efix
$\Gamma, \mathbf{x} \div A \vdash_{\mathsf{s}} F^\star \div A \mathbin{@} \omega$ by induction hypothesis

$$\begin{array}{ll} \quad \Gamma, x_p : \bigcirc A, \mathbf{x} \div A \vdash_{\mathsf{s}} F^\star \div A \;@\; \omega & \text{by weakening} \\ \Gamma, x_p : \bigcirc A \vdash_{\mathsf{s}} \mathsf{letcmp}\; y_v \lhd x_p \;\mathsf{in}\; y_v \div A \;@\; \omega & \text{(typing derivation)} \\ \Gamma, x_p : \bigcirc A \vdash_{\mathsf{s}} [\mathsf{letcmp}\; y_v \lhd x_p \;\mathsf{in}\; y_v/\mathbf{x}]F^\star \div A \;@\; \omega & \text{by Lemma 2.16} \\ \quad \Gamma \vdash_{\mathsf{s}} \mathsf{letcmp}\; y_r \lhd \mathsf{fix}\; x_p{:}\bigcirc A.\, \mathsf{cmp}\; [\mathsf{letcmp}\; y_v \lhd x_p \;\mathsf{in}\; y_v/\mathbf{x}]F^\star \;\mathsf{in}\; y_r \div A \;@\; \omega & \\ & \text{(typing derivation)} \\ \quad \Gamma \vdash_{\mathsf{s}} (\mathsf{efix}\; \mathbf{x}{\div}A.\, F)^\star \div A \;@\; \omega & \text{by the definition of } (\cdot)^\star \end{array}$$

□

Since $M^\star$ and $E^\star$ do not contain expression fixed point constructs, the rule Efix is not used in $\Gamma \vdash_{\mathsf{s}} M^\star : A \;@\; \omega$ and $\Gamma \vdash_{\mathsf{s}} E^\star \div A \;@\; \omega$. Neither is the rule Evar used unless M or E contains free expression variables. Therefore, given a term or expression with no free expression variable, the function $(\cdot)^\star$ returns another term or expression of the same type which does not need the rules Evar and Efix.

Propositions 2.22 and 2.23 show that the reduction rule *Efix* is sound and complete with respect to the operational semantics (in the direct style) in Section 2.3.5. We use the fact that the computation of $E^\star$ does not require the rule *Efix*.

Proposition 2.18.
For any term N, we have $([N/x]M)^\star = [N^\star/x]M^\star$ and $([N/x]E)^\star = [N^\star/x]E^\star$.
For any expression F, we have $([F/\mathbf{x}]M)^\star = [F^\star/\mathbf{x}]M^\star$ and $([F/\mathbf{x}]E)^\star = [F^\star/\mathbf{x}]E^\star$.

Proof. By simultaneous induction on the structure of M and E. □

Lemma 2.19. *If $M \mapsto_{\mathsf{t}} N$, then $M^\star \mapsto_{\mathsf{t}} N^\star$.*

Proof. By induction on the structure of the derivation of $M \mapsto_{\mathsf{t}} N$. □

Lemma 2.20.
If $M^\star \mapsto_{\mathsf{t}} N'$, then there exists N such that $N' = N^\star$ and $M \mapsto_{\mathsf{t}} N$.

Proof. By induction on the structure of the derivation of $M^\star \mapsto_{\mathsf{t}} N'$. □

We introduce an equivalence relation $\equiv_{\mathsf{e}}$ on expressions to state that two expressions compute to the same value.

Definition 2.21.
$E \equiv_{\mathsf{e}} F$ if and only if $E \;@\; \omega \mapsto_{\mathsf{e}}^* V \;@\; \omega'$ implies $F \;@\; \omega \mapsto_{\mathsf{e}}^* V \;@\; \omega'$, and vice versa.

The following equivalences are used in proofs below:

$$\begin{array}{rcll} \mathsf{letcmp}\; x \lhd \mathsf{cmp}\; E \;\mathsf{in}\; x & \equiv_{\mathsf{e}} & E & \\ \mathsf{letcmp}\; x \lhd \mathsf{cmp}\; E \;\mathsf{in}\; F & \equiv_{\mathsf{e}} & \mathsf{letcmp}\; x \lhd \mathsf{cmp}\; E' \;\mathsf{in}\; F \quad \textit{where} \quad E \equiv_{\mathsf{e}} E' \\ (\mathsf{efix}\; \mathbf{x}{\div}A.\, E)^\star & \equiv_{\mathsf{e}} & [(\mathsf{efix}\; \mathbf{x}{\div}A.\, E)^\star/\mathbf{x}]E^\star & \end{array}$$

The third equivalence follows from an expression reduction

$$(\mathsf{efix}\; \mathbf{x}{\div}A.\, E)^\star \;@\; \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\; y_r \lhd \mathsf{cmp}\; [(\mathsf{efix}\; \mathbf{x}{\div}A.\, E)^\star/\mathbf{x}]E^\star \;\mathsf{in}\; y_r \;@\; \omega.$$

Proposition 2.22.
If $E \;@\; \omega \mapsto_{\mathsf{e}} F \;@\; \omega'$ with the rule Efix, then $E^\star \;@\; \omega \mapsto_{\mathsf{e}} F' \;@\; \omega'$ and $F' \equiv_{\mathsf{e}} F^\star$.

Proof. By induction on the structure of the derivation of $E \ @ \ \omega \mapsto_{\mathsf{e}} F \ @ \ \omega'$. We consider the case $E =$ $\mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E_0$ where $M \neq \mathsf{cmp}\ E'$.

If $\mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E_0 \ @ \ \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0 \ @ \ \omega$ by the rule E_{Bind}, then $M \mapsto_{\mathsf{t}} N$.

By Lemma 2.19, $M^\star \mapsto_{\mathsf{t}} N^\star$.

Since $(\mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E_0)^\star = \mathsf{letcmp}\ x \triangleleft M^\star \ \mathsf{in}\ {E_0}^\star$ and $(\mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0)^\star = \mathsf{letcmp}\ x \triangleleft N^\star \ \mathsf{in}\ {E_0}^\star$, we have $(\mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E_0)^\star \ @ \ \omega \mapsto_{\mathsf{e}} (\mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0)^\star \ @ \ \omega$.

Then we let $F' = (\mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0)^\star$. □

Proposition 2.23.

If $E^\star \ @ \ \omega \mapsto_{\mathsf{e}} F' \ @ \ \omega'$, then there exists F such that $F' \equiv_{\mathsf{e}} F^\star$ and $E \ @ \ \omega \mapsto_{\mathsf{e}} F \ @ \ \omega'$.

Proof. By induction on the structure of the derivation of $E^\star \ @ \ \omega \mapsto_{\mathsf{e}} F' \ @ \ \omega'$. An interesting case is when the rule E_{Bind} is applied last in a given derivation.
If $E = \mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E_0$, then $E^\star = \mathsf{letcmp}\ x \triangleleft M^\star \ \mathsf{in}\ {E_0}^\star$.

By Lemma 2.20, there exists N such that $M \mapsto_{\mathsf{t}} N$ and $M^\star \mapsto_{\mathsf{t}} N^\star$.

Hence we have $E \ @ \ \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0 \ @ \ \omega'$ and $E^\star \ @ \ \omega \mapsto_{\mathsf{e}} \mathsf{letcmp}\ x \triangleleft N^\star \ \mathsf{in}\ {E_0}^\star \ @ \ \omega'$ (where $\omega = \omega'$).

Then we let $F = \mathsf{letcmp}\ x \triangleleft N \ \mathsf{in}\ E_0$.
If $E = \mathsf{efix}\ \mathbf{x} \div A.\ E_0$, then $F' \equiv_{\mathsf{e}} ([\mathsf{efix}\ \mathbf{x} \div A.\ E_0/\mathbf{x}]E_0)^\star$ (and $\omega = \omega'$)

because $(\mathsf{efix}\ \mathbf{x} \div A.\ E_0)^\star \equiv_{\mathsf{e}} [(\mathsf{efix}\ \mathbf{x} \div A.\ E_0)^\star/\mathbf{x}]{E_0}^\star = ([\mathsf{efix}\ \mathbf{x} \div A.\ E_0/\mathbf{x}]E_0)^\star$.

Then we let $F = [\mathsf{efix}\ \mathbf{x} \div A.\ E_0/\mathbf{x}]E_0$. □

As seen in the definition of expression fixed point constructs, term fixed point constructs can leak into expressions to give rise to recursive computations. Note that non-terminating computations in $\lambda_\bigcirc$ are not necessarily due to (term or expression) fixed point constructs, since mutable references can also be exploited to encode recursive computations. For example, the following expression initiates a non-terminating computation in which reference x stores a computation term which dereferences itself:

$$\left.\begin{array}{l}
\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ \mathsf{new}\ \mathsf{cmp}\ 0\ \mathsf{in} \\
\mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{write}\ x\ \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ x\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} \\
\quad \mathsf{in} \\
\mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\
z
\end{array}\right] @\ \cdot$$

$$\mapsto_{\mathsf{e}}^{*}\ \left.\begin{array}{l}
\mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{write}\ l\ \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ l\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} \\
\quad \mathsf{in} \\
\mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\
z
\end{array}\right] @\ [l \mapsto \mathsf{cmp}\ 0 : \bigcirc\ \mathsf{int}]$$

$$\mapsto_{\mathsf{e}}^{*}\ \left.\begin{array}{l}
\mathsf{letcmp}\ z \triangleleft \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ l\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} \\
\quad \mathsf{in} \\
z
\end{array}\right] @\ [l \mapsto \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ l\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} : \bigcirc\ \mathsf{int}]$$

$$\mapsto_{\mathsf{e}}^{*}\ \left.\begin{array}{l}
\mathsf{letcmp}\ z \triangleleft \\
\quad \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ z \triangleleft \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ l\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} \\ \qquad\quad \mathsf{in} \\ z\ \ ) \end{array} \\
\quad \mathsf{in} \\
z
\end{array}\right] @\ [l \mapsto \mathsf{cmp}\ (\ \begin{array}[t]{l} \mathsf{letcmp}\ y \triangleleft \mathsf{cmp}\ \mathsf{read}\ l\ \mathsf{in} \\ \mathsf{letcmp}\ z \triangleleft y\ \mathsf{in} \\ z\ \ ) \end{array} : \bigcirc\ \mathsf{int}]$$

$$\mapsto_{\mathsf{e}}^{*}\ \cdots$$

2.5.2 Backpatching semantics

Unlike the unfolding semantics, the backpatching semantics evaluates or computes a fixed point construct by first finishing the reduction of its body and then *"tying a recursive knot"*, or *"backpatching"* the result. For term evaluations, the two semantics are equivalent except that when the unfolding semantics gives rise to an infinite loop, the backpatching semantics generates an error.

We investigate a fixed point construct $\mathsf{vfix}\ \underline{z}\!:\!A.\,E$ for expressions that is based upon the backpatching semantics. Unlike $\mathsf{efix}\ \mathbf{x}\div A.\,E$ which computes a fixed point over both values and world effects and thus $\mathbf{x}$ is interpreted as an expression, it computes a fixed point only over values and $\underline{z}$ in it is a term.[5] For this reason, the computation is usually referred to as *value recursion* [18]. Similar constructs are found in Erkök and Launchbury [18] (fixed point construct mfix in Haskell) and Launchbury and Peyton Jones [37] (recursive state transformer fixST in Haskell).

[5] In this regard, the two fixed point constructs for expressions cannot be compare directly.

Syntax and type system

We introduce a *recursion variable* $\underline{z}$ (with an underscore) as a term and a *value recursion construct* $\mathsf{vfix}\,\underline{z}\!:\!A.\,E$ as an expression:

$$\begin{array}{lll} \text{term} & M & ::= \;\; \cdots \mid \underline{z} \\ \text{expression} & E & ::= \;\; \cdots \mid \mathsf{vfix}\,\underline{z}\!:\!A.\,E \end{array}$$

A substitution for $\underline{z}$ is defined in a standard way. To simplify the presentation of the type preservation theorem (Theorem 2.25), we separate recursion variables from ordinary variables in the type system by introducing a *value recursion context* Σ for recursion variables:

$$\text{value recursion context} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, \underline{z} : A$$

A typing judgment now includes a value recursion context to record the type of each recursion variable:

$$\begin{array}{ll} \text{term typing judgment} & \Gamma; \Sigma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega \\ \text{expression typing judgment} & \Gamma; \Sigma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega \end{array}$$

Typing rules for judgments $\Gamma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ and $\Gamma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ induce those for judgments $\Gamma; \Sigma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ and $\Gamma; \Sigma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ in a straightforward way (by adding Σ to every judgment). We also need additional rules for recursion variables and value recursion constructs:

$$\frac{}{\Gamma; \Sigma, \underline{z} : A \vdash_{\mathsf{s}} \underline{z} : A \mathbin{@} \omega}\ \mathsf{Vvar} \qquad \frac{\Gamma; \Sigma, \underline{z} : A \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega}{\Gamma; \Sigma \vdash_{\mathsf{s}} \mathsf{vfix}\,\underline{z}\!:\!A.\,E \div A \mathbin{@} \omega}\ \mathsf{Vfix}$$

The monotonicity of the accessibility relation $\leq$ (in Proposition 2.7) is now stated with new typing judgments.

Proposition 2.24.
If $\omega \leq \omega'$, *then*
$\Gamma; \Sigma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$ *implies* $\Gamma; \Sigma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega'$, *and*
$\Gamma; \Sigma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$ *implies* $\Gamma; \Sigma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega'$.

Operational semantics

Conceptually we compute $\mathsf{vfix}\,\underline{z}\!:\!A.\,E$ as follows: first we bind $\underline{z}$ to a *black hole* so that any premature attempt to read it results in a *value recursion error*; next we compute E to obtain a value V; finally we "backpatch" every occurrence of $\underline{z}$ in V with V itself and return the backpatched value as the result.

One approach to backpatching $\underline{z}$ with V is by replacing $\underline{z}$ by a fixed point construct $\mathsf{fix}\,\underline{z}\!:\!A.\,V$ (as in [47]). A problem with this approach is that $\underline{z}$ may appear at the resultant world after computing E. That is, if E at a world ω computes to V at another world ω', $\underline{z}$ may be used by ω'. Then we would need substitutions on worlds as well (*e.g.*, $[\mathsf{fix}\,\underline{z}\!:\!A.\,V/\underline{z}]\omega'$), which should be defined for each kind of world effect and thus we want to avoid; besides the type preservation property becomes difficult to prove.

To eliminate the need for substitutions on worlds, we maintain a *recursion store* σ. It associates each recursion variable with a value V:

$$\text{recursion store} \quad \sigma \quad ::= \quad \cdot \mid \sigma, \underline{z} = V$$

Now we reformulate the operational semantics with two reduction judgments:

- A term reduction $M \,\centerdot\, \sigma \mapsto_{\mathsf{t}} N$ means that M with recursion store σ reduces to N.

- An expression reduction $E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \mathbin{.} \sigma'$ means that E at world ω with recursion store σ reduces to F at world ω' with recursion store σ'.

A term reduction requires (but does not update) a recursion store because it may read recursion variables. An expression reduction may update both a world (by reducing instructions) and a recursion store (by reducing value recursion constructs). Reduction rules for judgments $M \mapsto_{\mathsf{t}} N$ and $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$ induce those for judgments $M \mathbin{.} \sigma \mapsto_{\mathsf{t}} N$ and $E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \mathbin{.} \sigma'$ in a straightforward way (by adding σ to every judgment).

Instead of directly modeling black holes with certain special values, we indirectly model black holes by reducing $\mathsf{vfix}\ \underline{z}\!:\!A.\,E$ to an intermediate value recursion construct $\mathsf{vfix}_{\bullet}\ \underline{z}:A.\,E$. That is, the presence of $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$ means that $\underline{z}$ is assumed to be bound to a black hole and that E is currently being reduced; if a term in E attempts to read $\underline{z}$, it results in a value recursion error and the whole reduction gets stuck. The typing rule for $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$ is the same as for $\mathsf{vfix}\ \underline{z}\!:\!A.\,E$:

$$\text{expression} \quad E \quad ::= \quad \cdots \mid \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$$

$$\frac{\Gamma; \Sigma, \underline{z} : A \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega}{\Gamma; \Sigma \vdash_{\mathsf{s}} \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E \div A \mathbin{@} \omega}\ \mathsf{Vfix}_{\bullet}$$

The rules for reducing recursion variables and value recursion constructs are as follows:

$$\frac{\underline{z} = V \in \sigma}{\underline{z} \mathbin{.} \sigma \mapsto_{\mathsf{t}} V}\ \mathit{Vvar}$$

$$\frac{}{\mathsf{vfix}\ \underline{z}\!:\!A.\,E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E \mathbin{@} \omega \mathbin{.} \sigma}\ \mathit{Vfix}_{init}$$

$$\frac{E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \mathbin{.} \sigma'}{\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,F \mathbin{@} \omega' \mathbin{.} \sigma'}\ \mathit{Vfix}_{red}$$

$$\frac{\underline{z} = V' \notin \sigma}{\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,V \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} V \mathbin{@} \omega \mathbin{.} \sigma, \underline{z} = V}\ \mathit{Vfix}_{bpatch}$$

These rules ensure that any premature attempt to read a recursion variable bound to a black hole results in a value recursion error and the whole reduction gets stuck. The rule Vvar implies that $\underline{z}$ is not a value in itself. The rule Vfix_{init} initiates the computation of $\mathsf{vfix}\ \underline{z}\!:\!A.\,E$ by reducing it to $\mathsf{vfix}_{\bullet}\ \underline{z} : A.\,E$; the rule Vfix_{red} reduces the body E of $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$; the rule Vfix_{bpatch} backpatches $\underline{z}$ with V. Note that α-conversion is freely applicable even to $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$.

The reduction rule Vfix_{bpatch} assumes *dynamic renaming* of recursion variables so that all recursion variables in a recursion store remain distinct. As an example, consider the following expression:

$$\mathsf{letcmp}\ x_1 \lhd \mathsf{cmp}\ \mathsf{vfix}\ \underline{z}\!:\!A.\,E_1\ \mathsf{in}\ \mathsf{letcmp}\ x_2 \lhd \mathsf{cmp}\ \mathsf{vfix}\ \underline{z}\!:\!A.\,E_2\ \mathsf{in}\ F$$

Although we do not need to rename either instance of $\underline{z}$ during typechecking, we have to rename the second instance after computing $\mathsf{vfix}\ \underline{z}\!:\!A.\,E_2$ because the recursion store already contains a recursion variable of the same name.

Since the result of an evaluation or a computation may contain recursion variables, we need to incorporate recursion stores or their abstractions in stating the type preservation property. We use value recursion contexts for this purpose as they are essentially the result of typing recursion stores. Formally we write $\models \sigma : \Sigma \mathbin{@} \omega$ if there exists a one-to-one correspondence between $\underline{z} = V \in \sigma$ and $\underline{z} : A \in \Sigma$ such that $\cdot; \Sigma \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$ holds. Now type preservation property is stated as follows:

Theorem 2.25 (Type preservation). *Suppose* $\models \sigma : \Sigma \mathbin{@} \omega$.

If $M \mathbin{.} \sigma \mapsto_{\mathsf{t}} N$ *and* $\cdot; \Sigma \vdash_{\mathsf{s}} M : A \mathbin{@} \omega$, *then* $\cdot; \Sigma \vdash_{\mathsf{s}} N : A \mathbin{@} \omega$.

If $E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \mathbin{.} \sigma'$ *and* $\cdot; \Sigma \vdash_{\mathsf{s}} E \div A \mathbin{@} \omega$, *then there exists* Σ' *such that* $\cdot; \Sigma' \vdash_{\mathsf{s}} F \div A \mathbin{@} \omega'$ *and* $\models \sigma' : \Sigma' \mathbin{@} \omega'$.

Proof. By induction on the structure of the derivation of $M \mathbin{.} \sigma \mapsto_{\mathsf{t}} N$ and $E \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \mathbin{.} \sigma'$. Interesting cases are when one of the rules $Vvar$, $Vfix_{init}$, $Vfix_{red}$, and $Vfix_{bpatch}$ is applied last in a given derivation. We consider two representative cases below.

$$Case\quad \frac{\underline{z} = V \in \sigma}{\underline{z} \mathbin{.} \sigma \mapsto_{\mathsf{t}} V}\; Vvar\ :$$

$\cdot; \Sigma \vdash_{\mathsf{s}} \underline{z} : A \mathbin{@} \omega$ implies $\underline{z} : A \in \Sigma$ by the rule Vvar.

From $\models \sigma : \Sigma \mathbin{@} \omega$, $\underline{z} = V \in \sigma$, and $\underline{z} : A \in \Sigma$,

we have $\cdot; \Sigma \vdash_{\mathsf{s}} V : A \mathbin{@} \omega$.

$$Case\quad \frac{\underline{z} = V' \notin \sigma}{\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\, V \mathbin{@} \omega \mathbin{.} \sigma \mapsto_{\mathsf{e}} V \mathbin{@} \omega \mathbin{.} \sigma, \underline{z} = V}\; Vfix_{bpatch}$$

Since $\models \sigma : \Sigma \mathbin{@} \omega$,

for any $\underline{z'} = V' \in \sigma$, we have $\cdot; \Sigma \vdash_{\mathsf{s}} V' \div A' \mathbin{@} \omega$ and $\underline{z'} : A' \in \Sigma$ for some type A'.

We let $\Sigma' = \Sigma, \underline{z} : A$.

Then, for any $\underline{z'} = V' \in \sigma$, we have $\cdot; \Sigma' \vdash_{\mathsf{s}} V' \div A' \mathbin{@} \omega$ and $\underline{z'} : A' \in \Sigma'$ for some type A'.

The rule $\mathsf{Vfix}_{\bullet}$ implies $\cdot; \Sigma \vdash_{\mathsf{s}} \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\, V \div A \mathbin{@} \omega$ and $\cdot; \Sigma, \underline{z} : A \vdash_{\mathsf{s}} V \div A \mathbin{@} \omega$.

Then $\cdot; \Sigma' \vdash_{\mathsf{s}} V \div A \mathbin{@} \omega$ and $\underline{z} : A \in \Sigma'$.

Therefore $\models \sigma, \underline{z} = V : \Sigma' \mathbin{@} \omega$. □

Since the type system does not detect value recursion errors, the computation of a well-typed expression may end up with a value recursion error. To catch value recursion errors statically, we can adopt advanced type systems for value recursion in [9, 16].

Simulating value recursion constructs

Section 2.5.1 has shown that $\mathsf{efix}\ \mathbf{x} \div A.\, E$ can be simulated with $\mathsf{fix}\, x\!:\!A.\, M$. Can we also simulate $\mathsf{vfix}\ \underline{z}\!:\!A.\, E$ with $\mathsf{fix}\, x\!:\!A.\, M$? In Haskell, a value recursion construct mfix for a specific monad can be defined in terms of the ordinary fixed point construct fix. For example, Moggi and Sabry [47] show that for a state monad $\mathsf{M}\ A = S \to (A \times S)$ where M is a type constructor and S is the type of states, mfix can be defined as follows:

$$\mathsf{mfix}\, x\!:\!A.\, M \ = \ \lambda s\!:\!S.\, \mathsf{fix}\, p\!:\!A \times S.\, (\lambda x\!:\!A.\, M)\ (\mathsf{fst}\ p)\ s$$

Here we use a product type $A \times S$ and a projection term $\mathsf{fst}\ p$; both M and $\mathsf{mfix}\ x : A.\, M$ have type $\mathsf{M}\ A = S \to (A \times S)$. Since the type constructor $\bigcirc$ in $\lambda_{\bigcirc}$ essentially forms a state monad, it may appear that we can define $\mathsf{vfix}\ \underline{z}\!:\!A.\, E$ in terms of $\mathsf{fix}\, x\!:\!A.\, M$. Unlike the state monad $\mathsf{M}\ A$, however, we cannot access states (*i.e.*, worlds) as terms. Therefore we cannot exploit the above idea to simulate $\mathsf{vfix}\ \underline{z}\!:\!A.\, E$ with $\mathsf{fix}\, x\!:\!A.\, M$.

Another idea to simulate $\mathsf{vfix}\ \underline{z}\!:\!A.\, E$ is to use instructions for mutable references: to compute $\mathsf{vfix}\ \underline{z}\!:\!A.\, E$, we initialize a fresh reference for $\underline{z}$; to backpatch $\underline{z}$, we update the store. In this case, $\underline{z}$ can no longer be a term because its evaluation requires an access to the store. In other words, $\underline{z}$ should now be defined as an expression.

$$\begin{array}{llll}
\text{term} & M & ::= & \cdots \mid \mathsf{cont_t}\ \kappa \mid \mathsf{callcc_t}\ x.\, M \mid \mathsf{throw_t}\ M\ M \\
\text{value} & V & ::= & \cdots \mid \mathsf{cont_t}\ \kappa \\
\text{evaluation context} & \kappa & ::= & [] \mid \kappa\ M \mid (\lambda x\!:\!A.\, M)\ \kappa \mid \mathsf{throw_t}\ \kappa\ M \mid \mathsf{throw_t}\ (\mathsf{cont_t}\ \kappa)\ \kappa
\end{array}$$

Figure 2.8: Syntax for continuations for terms.

2.6 Continuations

So far, we have restricted ourselves to world effects, *i.e.*, transitions between worlds. $\lambda_\bigcirc$ confines world effects to expressions so that terms are free of world effects. When we extend $\lambda_\bigcirc$ with control effects, however, it is not immediately clear which syntactic category should be permitted to produce control effects. On one hand, we could choose to confine control effects to expressions so that terms remain free of any kind of effect. Then the distinction between effect-free evaluations and effectful computations is drawn in a conventional sense. On the other hand, in order to develop $\lambda_\bigcirc$ into a practical programming language, it is desirable to allow control effects in terms. For example, exceptions for terms would be an easy way to handle division by zero or pattern-match failures occurring during evaluations. At the same time, however, exceptions for expressions are also useful for those instructions whose execution does not always succeed.

We hold the view that expressions are in principle a syntactic category specialized for world effects, and allow control effects in *both terms and expressions*. The decision does not prevent us from developing control effects orthogonally to world effects, since control effects are realized with reduction rules whereas world effects are realized with world structures. In fact, there is no reason to confine control effects only to one syntactic category, since the concept of control effect is relative to what constitutes the "basic" reduction rules anyway.

As an example of control effect, we consider continuations. We consider two kinds: one for terms and another for expressions. A continuation for terms denotes an evaluation parameterized over terms; a continuation for expressions denotes a computation parameterized over terms. The two are independent notions, and we discuss them separately. Since we are primarily interested in how continuations change the state of the run-time system, we focus on the operational semantics only; for the type system, we refer the reader to the literature (*e.g.*, [25]).

In the syntax, we assume value recursion constructs which interact with continuations for expressions in an interesting way. Hence we continue to use the two reduction judgments $M \,\centerdot\, \sigma \mapsto_{\mathsf{t}} N$ and $E \mathbin{@} \omega \,\centerdot\, \sigma \mapsto_{\mathsf{e}} F \mathbin{@} \omega' \,\centerdot\, \sigma'$ in Section 2.5.2 (but in a different style).

2.6.1 Continuations for terms

Figure 2.8 shows the syntax for continuations for terms. An *evaluation context* κ is a term with a hole $[]$ which can be filled with a term M to produce another term $\kappa[M]$; it assumes a call-by-value discipline. $\mathsf{cont_t}\ \kappa$ lifts an evaluation context κ to a value and is called a *term continuation*. $\mathsf{callcc_t}$ and $\mathsf{throw_t}$ are constructs for capturing and throwing term continuations, respectively.

The operational semantics in Figure 2.9 uses a reduction judgment in the form of $\kappa[M] \,\centerdot\, \sigma \mapsto_{\mathsf{t}} \kappa'[N]$ where σ is a recursion store. Note that it is the same term reduction judgment as in Section 2.5.2 because both $\kappa[M]$ and $\kappa'[N]$ are terms. The rule CTred uses a term reduction $M \Rightarrow_{\beta\mathsf{term}} N$. The rule $\mathit{CTcallcc}$ binds variable x to a term continuation containing the current evaluation context κ; the rule $\mathit{CTthrow}$ nullifies the current evaluation context κ to activate a new evaluation context κ'.

The formulation of continuations for terms is standard. What is interesting is that from a logical perspective, continuations for terms change the meaning of $A\ \mathit{true}$ from intuitionistic truth to classical truth [23]. The change in the meaning of $A\ \mathit{true}$, however, does not mean that we have to change the definition of

$$\frac{M \;\Rightarrow_{\beta\mathsf{term}}\; N}{\kappa[M]_{\,\centerdot}\,\sigma \mapsto_{\mathsf{t}} \kappa[N]}\ \mathit{CTred} \qquad \frac{\underline{z} = V \in \sigma}{\kappa[\underline{z}]_{\,\centerdot}\,\sigma \mapsto_{\mathsf{t}} \kappa[V]}\ \mathit{CTvvar}$$

$$\frac{}{\kappa[\mathsf{callcc_t}\ x.\,M]_{\,\centerdot}\,\sigma \mapsto_{\mathsf{t}} \kappa[[\mathsf{cont_t}\ \kappa/x]M]}\ \mathit{CTcallcc}$$

$$\frac{}{\kappa[\mathsf{throw_t}\ (\mathsf{cont_t}\ \kappa')\ V]_{\,\centerdot}\,\sigma \mapsto_{\mathsf{t}} \kappa'[V]}\ \mathit{CTthrow}$$

Figure 2.9: Reduction rules for continuations for terms.

$$\begin{array}{llll}
\text{term} & M & ::= & \cdots \mid \mathsf{cont_e}\ \phi \\
\text{value} & V & ::= & \cdots \mid \mathsf{cont_e}\ \phi \\
\text{expression} & E & ::= & \cdots \mid \mathsf{callcc_e}\ x.\,E \mid \mathsf{throw_e}\ M\ E \\
\text{computation context} & \phi & ::= & []_{\mathsf{e}} \mid []_{\mathsf{t}} \mid \mathsf{letcmp}\ x \triangleleft []_{\mathsf{t}}\ \mathsf{in}\ E \mid \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ \phi\ \mathsf{in}\ E \mid \\
 & & & \mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,\phi \mid \mathsf{throw_e}\ []_{\mathsf{t}}\ E \mid \mathsf{throw_e}\ (\mathsf{cont_e}\ \phi)\ \phi
\end{array}$$

Figure 2.10: Syntax for continuations for expressions.

expressions accordingly, since our definition of $A\ \mathit{comp}$ is not subject to a particular definition of $A\ \mathit{true}$. In other words, even if we change the meaning of $A\ \mathit{true}$, the same definition of $A\ \mathit{comp}$ remains valid with respect to the new definition of $A\ \mathit{true}$; hence the previous definition of expressions also remains valid.

2.6.2 Continuations for expressions

Figure 2.10 shows the syntax for continuations for expressions. A *computation context* ϕ is an expression with a hole $[]_{\mathsf{t}}$ or $[]_{\mathsf{e}}$. $[]_{\mathsf{t}}$ can be filled only with a term, and $[]_{\mathsf{e}}$ only with an expression. $\mathsf{cont_e}\ \phi$ lifts a computation context ϕ to a value and is called an *expression continuation*. $\mathsf{callcc_e}$ and $\mathsf{throw_e}$ are constructs for capturing and throwing expression continuations, respectively.

The operational semantics in Figure 2.11 uses a reduction judgment in the form of $\phi[E]\ @\ \omega\,_{\centerdot}\,\sigma \mapsto_{\mathsf{e}} \phi'[F]\ @\ \omega'\,_{\centerdot}\,\sigma'$. Note that it is the same expression reduction judgment as in Section 2.5.2 because both $\phi[E]$ and $\phi'[F]$ are expressions. The rule *CEcallcc* binds variable x to a expression continuation containing the current computation context ϕ; the rule *CEthrow* nullifies the current computation context ϕ to activate a new computation context ϕ'. By the rule *CEvfixo*, a computation context $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,\phi$ marks that $\underline{z}$ is bound to a black hole.

It is important that the rule *CEvfixc* does not require $\underline{z} = V' \notin \sigma$ in the premise; if $\underline{z} = V'$ is already in σ, it is removed in $\sigma, \underline{z} = V$ (so that all recursion variables remain distinct). The reason is that an expression continuation that has been captured *before* the completion of the computation of $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$ may be thrown *after* its completion. In this case, recursion variable $\underline{z}$ is *already* bound to the value that the previous computation of $\mathsf{vfix}_{\bullet}\ \underline{z}\!:\!A.\,E$ has returned. We can exploit this property to show that, for example, $\mathsf{vfix}\ \underline{z}\!:\!A.\ \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$ and $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ \mathsf{vfix}\ \underline{z}\!:\!A.\,E$ behave differently even when $\underline{z}$ is not free in M.[6]

Consider an expression

$$\mathsf{vfix}\ \underline{z}\!:\!A.\ \mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ \mathsf{callcc_e}\ y.\,E\ \mathsf{in}\ F$$

where $\underline{z}$ is not free in E. The expression continuation captured by $\mathsf{callcc_e}\ y.\,E$ may escape the scope of the whole value recursion construct. When it is thrown later, $\underline{z}$ is already bound to a value and every attempt to

[6] Erkök and Launchbury [18] call the equivalence between the two expressions the *left-shrinking* property of value recursion.

$$\frac{M \,.\, \sigma \mapsto_{\mathsf{t}} N}{\phi[M] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi[N] \;@\; \omega \,.\, \sigma} \; \mathit{CEtred}$$

$$\frac{}{\phi[\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ V\ \mathsf{in}\ E] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi[[V/x]E] \;@\; \omega \,.\, \sigma} \; \mathit{CEbind}$$

$$\frac{}{\phi[\mathsf{callcc_e}\ x.\, E] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi[[\mathsf{cont_e}\ \phi/x]E] \;@\; \omega \,.\, \sigma} \; \mathit{CEcallcc}$$

$$\frac{}{\phi[\mathsf{throw_e}\ (\mathsf{cont_e}\ \phi')\ V] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi'[V] \;@\; \omega \,.\, \sigma} \; \mathit{CEthrow}$$

$$\frac{}{\phi[\mathsf{vfix}\ \underline{z}\!:\!A.\, E] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi[\mathsf{vfix}_\bullet\ \underline{z}\!:\!A.\, E] \;@\; \omega \,.\, \sigma} \; \mathit{CEvfixo}$$

$$\frac{}{\phi[\mathsf{vfix}_\bullet\ \underline{z}\!:\!A.\, V] \;@\; \omega \,.\, \sigma \mapsto_{\mathsf{e}} \phi[V] \;@\; \omega \,.\, \sigma, \underline{z} = V} \; \mathit{CEvfixc}$$

Figure 2.11: Reduction rules for continuations for expressions.

read $\underline{z}$ in F succeeds without raising a value recursion error. This is not the case for the following expression:

$$\mathsf{letcmp}\ x \triangleleft \mathsf{cmp}\ \mathsf{callcc_e}\ y.\, E\ \mathsf{in}\ \mathsf{vfix}\ \underline{z}\!:\!A.\, F$$

During the computation of F, $\underline{z}$ is bound to a black hole by the rule $\mathit{CEvfixo}$. Consequently any attempt to read $\underline{z}$ in F results in a value recursion error.

In general, value recursion is unsafe in the presence of expression continuations because a value recursion construct may compute to a value containing *unresolved recursion variables*, that is, recursion variables bound to black holes (the counter-example in [47] can be rewritten in $\lambda_{\bigcirc}$). An error resulting from reading an unresolved recursion variable is similar to a value recursion error in that both result from an attempt to read a recursion variable bound to a black hole. The difference is that while a value recursion error results from a premature attempt to read a recursion variable that will be eventually bound to a value, an unresolved recursion variable remains bound to a black hole forever.

2.7 Summary

Moggi's monadic metalanguage λ_{ml} [44, 45] has served as the *de facto* standard for subsequent monadic languages [36, 37, 6, 70, 46, 78, 47]. Benton, Biermann, and de Paiva [7] show that from a type-theoretic perspective, λ_{ml} is connected to lax logic via the Curry-Howard isomorphism. Pfenning and Davies [60] reformulate λ_{ml} by applying Martin-Löf's methodology of distinguishing between propositions and judgments [42] to lax logic. The new formulation of λ_{ml} draws a syntactic distinction between values and computations, and uses the modality $\bigcirc$ for computations. It is used in the design of a security-typed monadic language [13]; its underlying modal type theory inspires type systems in [4, 5] and effect systems in [51, 52].

The idea of the syntactic distinction but without an explicit modality for computations is used by Petersen *et al.* [54]. The same idea is also used by Mandelbaum, Walker, and Harper [41]. Their language is similar to $\lambda_{\bigcirc}$ in that the operational semantics (but not the type system) uses an accessibility relation between worlds. The meaning of a world is, however, slightly different: a world in their language is a collection of facts on a world in $\lambda_{\bigcirc}$.

$\lambda_{\bigcirc}$ extends the new formulation of λ_{ml} by Pfenning and Davies with an operational semantics to support concrete notions of computational effect. Compared with those monadic languages based upon λ_{ml}, it does not strictly increase the expressive power — it is straightforward to devise a translation from $\lambda_{\bigcirc}$ to a typical monadic language based upon λ_{ml} and vice versa. In this regard, the syntactic distinction in $\lambda_{\bigcirc}$ may be thought of as a cosmetic change to the syntax of monadic languages. It, however, inspires a new approach to incorporating computational effects into monadic languages by allowing control effects both in terms and in expressions while confining world effects to expressions. In a monadic language based upon λ_{ml}, this (unorthodox) approach would mean that its pure functional sublanguage is allowed to produce control effects. The syntactic distinction also leads to the interpretation of terms and expressions as complete languages of their own, which makes $\lambda_{\bigcirc}$ a candidate for a unified framework under which to study two languages that have traditionally been studied separately: Haskell (corresponding to terms) and ML (corresponding to expressions). Ultimately we believe that the idea of the syntactic distinction conveys a design principle not found in other monadic languages.

Chapter 3

The Probabilistic Language PTP

This chapter presents the syntax, type system, and operational semantics of PTP. We give examples to demonstrate properties of PTP, and show how to verify that a program correctly encodes a target probability distribution. We propose the Monte Carlo method [40] as a means of overcoming a limitation of PTP, namely lack of support for precise reasoning about probability distributions.

For the reader who has read the previous chapter, PTP may be viewed as a simplified account of $\lambda_{\bigcirc}$ with language constructs for probabilistic computations in Section 2.4.1. A source of simplification is that a world, which is an infinite sequence of random numbers, does not affect types of terms and expressions; hence typing judgments in PTP do not require worlds. The following table show judgments in $\lambda_{\bigcirc}$ and their corresponding judgments in PTP:

Judgments in $\lambda_{\bigcirc}$	Judgments in PTP
$\Gamma \vdash_{\mathsf{s}} M : A \ @ \ \omega$	$\Gamma \vdash_{\mathsf{p}} M : A$
$\Gamma \vdash_{\mathsf{s}} E \div A \ @ \ \omega$	$\Gamma \vdash_{\mathsf{p}} E \div A$
$M \mapsto_{\mathsf{t}} N$	(same)
$M \rightharpoonup V$	(same)
$E \ @ \ \omega \mapsto_{\mathsf{e}} F \ @ \ \omega'$	(same)
$E \ @ \ \omega \rightharpoondown V \ @ \ \omega'$	(same)

The syntax of PTP uses type constructors familiar from programming languages (rather than logic) and more specific keywords specialized to probability distributions:

Syntax of $\lambda_{\bigcirc}$	Syntax of PTP
$A \supset B$	$A \rightarrow B$
$A \wedge B$	$A \times B$
$\mathsf{cmp}\ E$	$\mathsf{prob}\ E$
$\mathsf{letcmp}\ x \triangleleft M \ \mathsf{in}\ E$	$\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$

The definition of PTP in this chapter is self-contained, but should be supplemented by the previous chapter for its logical foundation.

3.1 Definition of PTP

3.1.1 Syntax and type system

PTP augments the lambda calculus, consisting of *terms*, with a separate syntactic category, consisting of *expressions* in a monadic syntax. Terms denote regular values and expressions denote probabilistic computations. We say that a term *evaluates* to a value and an expression *computes* to a sample.

type	A, B	$::=$	$A \to A \mid A \times A \mid \bigcirc A \mid \mathsf{real}$
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\,M \mid M\ M \mid (M, M) \mid \mathsf{fst}\ M \mid$ $\mathsf{snd}\ M \mid \mathsf{fix}\ x\!:\!A.\,M \mid \mathsf{prob}\ E \mid r$
expression	E, F	$::=$	$M \mid \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \mid \mathcal{S}$
value/sample	V	$::=$	$\lambda x\!:\!A.\,M \mid (V, V) \mid \mathsf{prob}\ E \mid r$
real number	r		
sampling sequence	ω	$::=$	$r_1 r_2 \cdots r_i \cdots \quad$ *where* $r_i \in (0.0, 1.0]$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Figure 3.1: Abstract syntax for PTP.

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{p}} x : A}\ \mathsf{Hyp} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{p}} M : B}{\Gamma \vdash_{\mathsf{p}} \lambda x\!:\!A.\,M : A \to B}\ \mathsf{Lam}$$

$$\frac{\Gamma \vdash_{\mathsf{p}} M_1 : A \to B \quad \Gamma \vdash_{\mathsf{p}} M_2 : A}{\Gamma \vdash_{\mathsf{p}} M_1\ M_2 : B}\ \mathsf{App} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M_1 : A_1 \quad \Gamma \vdash_{\mathsf{p}} M_2 : A_2}{\Gamma \vdash_{\mathsf{p}} (M_1, M_2) : A_1 \times A_2}\ \mathsf{Prod}$$

$$\frac{\Gamma \vdash_{\mathsf{p}} M : A_1 \times A_2}{\Gamma \vdash_{\mathsf{p}} \mathsf{fst}\ M : A_1}\ \mathsf{Fst} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M : A_1 \times A_2}{\Gamma \vdash_{\mathsf{p}} \mathsf{snd}\ M : A_2}\ \mathsf{Snd}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{p}} M : A}{\Gamma \vdash_{\mathsf{p}} \mathsf{fix}\ x\!:\!A.\,M : A}\ \mathsf{Fix} \qquad \frac{\Gamma \vdash_{\mathsf{p}} E \div A}{\Gamma \vdash_{\mathsf{p}} \mathsf{prob}\ E : \bigcirc A}\ \mathsf{Prob} \qquad \frac{}{\Gamma \vdash_{\mathsf{p}} r : \mathsf{real}}\ \mathsf{Real}$$

$$\frac{\Gamma \vdash_{\mathsf{p}} M : A}{\Gamma \vdash_{\mathsf{p}} M \div A}\ \mathsf{Term} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M : \bigcirc A \quad \Gamma, x : A \vdash_{\mathsf{p}} E \div B}{\Gamma \vdash_{\mathsf{p}} \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \div B}\ \mathsf{Bind}$$

$$\frac{}{\Gamma \vdash_{\mathsf{p}} \mathcal{S} \div \mathsf{real}}\ \mathsf{Sampling}$$

Figure 3.2: Typing rules of PTP.

Figure 3.1 shows the abstract syntax for PTP. We use x for variables. $\lambda x\!:\!A.\,M$ is a lambda abstraction, and $M\ M$ is an application term. (M, M) is a product term, and $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$ are projection terms; we include these terms to support joint distributions. $\mathsf{fix}\ x\!:\!A.\,M$ is a fixed point construct for recursive evaluations. A *probability term* $\mathsf{prob}\ E$ encapsulates expression E; it is a first-class value denoting a probability distribution. r is a real number.

There are three kinds of expressions: term M, *bind expression* $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$, and *sampling expression* $\mathcal{S}$. As an expression, M returns (with probability 1) the result of evaluating M. $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$ sequences two probabilistic computations (if M evaluates to a probability term). $\mathcal{S}$ consumes a random number in a *sampling sequence*, an infinite sequence of random numbers drawn independently from $U(0.0, 1.0]$.

The type system employs two kinds of typing judgments:

- Term typing judgment $\Gamma \vdash_{\mathsf{p}} M : A$, meaning that M evaluates to a value of type A under typing context Γ.

- Expression typing judgment $\Gamma \vdash_{\mathsf{p}} E \div A$, meaning that E computes to a sample of type A under typing context Γ.

A typing context Γ is a set of bindings $x : A$. Figure 3.2 shows the typing rules of PTP. The rule Prob is the introduction rule for the type constructor $\bigcirc$; it means that type $\bigcirc A$ denotes probability distributions over type A. The rule Bind is the elimination rule for the type constructor $\bigcirc$. The rule Term means that

$$\frac{M \mapsto_{\mathsf{t}} M'}{M\ N \mapsto_{\mathsf{t}} M'\ N}\ T_{\beta_L} \qquad \frac{N \mapsto_{\mathsf{t}} N'}{(\lambda x\!:\!A.\,M)\ N \mapsto_{\mathsf{t}} (\lambda x\!:\!A.\,M)\ N'}\ T_{\beta_R}$$

$$\frac{}{(\lambda x\!:\!A.\,M)\ V \mapsto_{\mathsf{t}} [V/x]M}\ T_{\beta_V} \qquad \frac{M \mapsto_{\mathsf{t}} M'}{(M, N) \mapsto_{\mathsf{t}} (M', N)}\ T_{P_L}$$

$$\frac{N \mapsto_{\mathsf{t}} N'}{(V, N) \mapsto_{\mathsf{t}} (V, N')}\ T_{P_R} \qquad \frac{M \mapsto_{\mathsf{t}} N}{\mathsf{fst}\ M \mapsto_{\mathsf{t}} \mathsf{fst}\ N}\ T_{Fst} \qquad \frac{}{\mathsf{fst}\ (V, V') \mapsto_{\mathsf{t}} V}\ T_{Fst'}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{snd}\ M \mapsto_{\mathsf{t}} \mathsf{snd}\ N}\ T_{Snd} \qquad \frac{}{\mathsf{snd}\ (V, V') \mapsto_{\mathsf{t}} V'}\ T_{Snd'}$$

$$\frac{}{\mathsf{fix}\ x\!:\!A.\,M \mapsto_{\mathsf{t}} [\mathsf{fix}\ x\!:\!A.\,M/x]M}\ T_{Fix} \qquad \frac{M \mapsto_{\mathsf{t}} N}{M\ @\ \omega \mapsto_{\mathsf{e}} N\ @\ \omega}\ E_{Term}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F\ @\ \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ F\ @\ \omega}\ E_{Bind}$$

$$\frac{E\ @\ \omega \mapsto_{\mathsf{e}} E'\ @\ \omega'}{\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F\ @\ \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E'\ \mathsf{in}\ F\ @\ \omega'}\ E_{BindR}$$

$$\frac{}{\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ V\ \mathsf{in}\ F\ @\ \omega \mapsto_{\mathsf{e}} [V/x]F\ @\ \omega}\ E_{BindV}$$

$$\frac{}{\mathcal{S}\ @\ r\omega \mapsto_{\mathsf{e}} r\ @\ \omega}\ \mathit{Sampling}$$

Figure 3.3: Operational semantics of PTP.

every term converts into a probabilistic computation that involves no probabilistic choice. The rule Real shows that real is the type of real numbers. A sampling expression $\mathcal{S}$ has also type real, as shown in the rule Sampling, because it computes to a real number.

3.1.2 Operational semantics

Since PTP draws a syntactic distinction between regular values and probabilistic computations, its operational semantics needs two kinds of judgments:

- Term evaluation judgment $M \rightharpoonup V$, meaning that term M evaluates to value V.
- Expression computation judgment $E\ @\ \omega \rightharpoondown V\ @\ \omega'$, meaning that expression E with sampling sequence ω computes to sample V with remaining sampling sequence ω'. Conceptually $E\ @\ \omega \rightharpoondown V\ @\ \omega'$ consumes random numbers in $\omega - \omega'$. Properties of the consumed sequence $\omega - \omega'$ (*e.g.*, its length) are not directly observable.

For term evaluations, we introduce a term reduction $M \mapsto_{\mathsf{t}} N$ in a call-by-value discipline (we could equally choose call-by-name or call-by-need). We identify $M \mapsto_{\mathsf{t}}^{*} V$ with $M \rightharpoonup V$, where $\mapsto_{\mathsf{t}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{t}}$. For expression computations, we introduce an expression reduction $E\ @\ \omega \mapsto_{\mathsf{e}} F\ @\ \omega'$ such that $E\ @\ \omega \mapsto_{\mathsf{e}}^{*} V\ @\ \omega'$ is identified with $E\ @\ \omega \rightharpoondown V\ @\ \omega'$, where $\mapsto_{\mathsf{e}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{e}}$. Both reductions use capture-avoiding term substitutions $[M/x]N$ and $[M/x]E$ defined in a standard way, as in Section 2.3.3.

Figure 3.3 shows the reduction rules in the operational semantics of PTP. Expression reductions may invoke term reductions (*e.g.*, to reduce M in $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$). The rules E_{BindR} and E_{BindV} mean that given a bind expression $\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F$, we finish computing E before substituting a value for x in F. Note that like a term evaluation, an expression computation itself is deterministic; it is only when we vary sampling sequences that an expression exhibits probabilistic behavior.

An expression computation $E \mathbin{@} \omega \mapsto_{\mathsf{e}}^{*} V \mathbin{@} \omega'$ means that E takes a sampling sequence ω, consumes a finite prefix of ω in order, and returns a sample V with the remaining sampling sequence ω':

Proposition 3.1. *If $E \mathbin{@} \omega \mapsto_{\mathsf{e}}^{*} V \mathbin{@} \omega'$, then $\omega = r_1 r_2 \cdots r_n \omega'$ ($n \geq 0$) where*

$$E \mathbin{@} \omega \mapsto_{\mathsf{e}}^{*} \cdots \mapsto_{\mathsf{e}}^{*} E_i \mathbin{@} r_{i+1} \cdots r_n \omega' \mapsto_{\mathsf{e}}^{*} \cdots \mapsto_{\mathsf{e}}^{*} E_n \mathbin{@} \omega' \mapsto_{\mathsf{e}}^{*} V \mathbin{@} \omega'$$

for a sequence of expressions $E_1, \cdots, E_n$.

Thus an expression computation coincides with the operational description of a sampling function when applied to a sampling sequence, which implies that an expression represents a sampling function. (Here we use a generalized notion of sampling function mapping $(0.0, 1.0]^\infty$ to $A \times (0.0, 1.0]^\infty$ for a certain type A.)

The type safety of PTP consists of two properties: type preservation and progress. Their proofs are omitted as they are special cases of Theorems 2.8 and 2.10, except for $\mathcal{S}$ which satisfies the type-preservation and monotonicity requirements on instructions.

Theorem 3.2 (Type preservation).
If $M \mapsto_{\mathsf{t}} N$ and $\cdot \vdash_{\mathsf{p}} M : A$, then $\cdot \vdash_{\mathsf{p}} N : A$.
If $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$ and $\cdot \vdash_{\mathsf{p}} E \div A$, then $\cdot \vdash_{\mathsf{p}} F \div A$.

Theorem 3.3 (Progress).
*If $\cdot \vdash_{\mathsf{p}} M : A$, then either M is a value (*i.e.*, $M = V$), or there exists N such that $M \mapsto_{\mathsf{t}} N$.*
*If $\cdot \vdash_{\mathsf{p}} E \div A$, then either E is a sample (*i.e.*, $E = V$), or for any sampling sequence ω, there exist F and ω' such that $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$.*

3.1.3 Fixed point construct for expressions

In PTP, expressions describe non-recursive probabilistic computations. Since some probability distributions are defined in a recursive way (*e.g.*, geometric distributions), it is desirable to be able to describe recursive probabilistic computations as well. To this end, we introduce an *expression variable* $\mathbf{x}$ and an *expression fixed point construct* $\mathsf{efix}\ \mathbf{x}{\div}A.\,E$; a new form of binding $\mathbf{x} \div A$ for expression variables is used in typing contexts:

$$\begin{array}{llll} \text{expression} & E & ::= & \cdots \mid \mathbf{x} \mid \mathsf{efix}\ \mathbf{x}{\div}A.\,E \\ \text{typing context} & \Gamma & ::= & \cdots \mid \Gamma, \mathbf{x} \div A \end{array}$$

New typing rules and reduction rule are as follows:

$$\frac{}{\Gamma, \mathbf{x} \div A \vdash_{\mathsf{p}} \mathbf{x} \div A}\ \mathsf{Evar} \qquad \frac{\Gamma, \mathbf{x} \div A \vdash_{\mathsf{p}} E \div A}{\Gamma \vdash_{\mathsf{p}} \mathsf{efix}\ \mathbf{x}{\div}A.\,E \div A}\ \mathsf{Efix}$$

$$\frac{}{\mathsf{efix}\ \mathbf{x}{\div}A.\,E \mathbin{@} \omega \mapsto_{\mathsf{e}} [\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]E \mathbin{@} \omega}\ \mathit{Efix}$$

In the rule Efix, $[\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]E$ denotes a capture-avoiding substitution of $\mathsf{efix}\ \mathbf{x}{\div}A.\,E$ for expression variable $\mathbf{x}$.

Expression fixed point constructs are syntactic sugar as they can be simulated with fixed point constructs for terms. See Section 2.5.1 for details.

3.1.4 Distinguishing terms and expressions

The *syntactic* distinction between terms and expressions in PTP is optional in the sense that the grammar does not need to distinguish expressions as a separate non-terminal. On the other hand, the *semantic* distinction, both statically (in the form of term and expression typing judgments) and dynamically (in the form of evaluation and computation judgments) appears to be essential for a clean formulation of PTP.

PTP is a conservative extension of a conventional language because terms constitute a conventional language of their own. In essence, term evaluations are always deterministic and we need only terms when writing deterministic programs. As a separate syntactic category, expressions provide a framework for probabilistic computation that abstracts from the definition of terms. For example, the addition of a new term construct does not change the definition of expressions. When programming in PTP, therefore, the syntactic distinction between terms and expressions aids us in deciding which of deterministic evaluations and probabilistic computations we should focus on. In the next section, we show how to encode various probability distributions and further investigate properties of PTP.

3.2 Examples

When encoding a probability distribution in PTP, we naturally concentrate on a method of generating samples, rather than calculating the probability assigned to each event. If the probability distribution itself is defined in terms of a process of generating samples, we simply translate the definition. If, however, the probability distribution is defined in terms of a probability measure or an equivalent, we may not always derive a sampling function in a mechanical manner. Instead we have to exploit its unique properties to devise a sampling function.

Below we show examples of encoding various probability distributions in PTP. These examples demonstrate three properties of PTP: a unified representation scheme for probability distributions, rich expressiveness, and high versatility in encoding probability distributions. The sampling methods used in the examples are all found in simulation theory [10]. Thus PTP is a programming language in which sampling methods developed in simulation theory can be formally expressed in a fashion that is concise and readable while remaining as efficient as the originals.

We assume primitive types int and bool (with boolean values True and False), arithmetic and comparison operators, and a conditional term construct $\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2$. We also assume standard let-binding, recursive $\mathsf{let\ rec}$-binding, and pattern matching when it is convenient for the examples.[1] We use the following syntactic sugar for expressions:

$$\begin{array}{rcl} \mathsf{unprob}\ M & \equiv & \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ x \\ \mathsf{eif}\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 & \equiv & \mathsf{unprob}\ (\mathsf{if}\ M\ \mathsf{then}\ \mathsf{prob}\ E_1\ \mathsf{else}\ \mathsf{prob}\ E_2) \end{array}$$

$\mathsf{unprob}\ M$ chooses a sample from the probability distribution denoted by M (we choose the keyword unprob to suggest that it does the opposite of what prob does.) $\mathsf{eif}\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2$ branches to either E_1 or E_2 depending on the result of evaluating M.

[1] If type inference and polymorphism are ignored, let-binding and recursive $\mathsf{let\ rec}$-binding may be interpreted as follows, where $\_$ is a wildcard pattern for types:

$$\begin{array}{rcl} \mathsf{let}\ x = M\ \mathsf{in}\ N & \equiv & (\lambda x\!:\!\_.\,N)\ M \\ \mathsf{let\ rec}\ x = M\ \mathsf{in}\ N & \equiv & \mathsf{let}\ x = \mathsf{fix}\ x\!:\!\_.\,M\ \mathsf{in}\ N \end{array}$$

Unified representation scheme

PTP provides a unified representation scheme for probability distributions. While its type system distinguishes between different probability domains, its operational semantics does not distinguish between different kinds of probability distributions, such as discrete, continuous, or neither. We show an example for each case.

We encode a Bernoulli distribution over type bool with parameter p as follows:

$$\begin{array}{l}\mathsf{let}\ bernoulli = \lambda p\!:\!\mathsf{real}.\ \mathsf{prob}\ \ \begin{array}[t]{l}\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\\ x \leq p\end{array}\end{array}$$

$bernoulli$ can be thought of as a binary choice construct. It is expressive enough to specify any discrete distribution with finite support. In fact, $bernoulli\ 0.5$ suffices to specify all such probability distributions, since it is capable of simulating a binary choice construct [21] (if the probability assigned to each element in the domain is computable).

As an example of continuous distribution, we encode a uniform distribution over a real interval $(a, b]$ by exploiting the definition of the sampling expression:

$$\begin{array}{l}\mathsf{let}\ uniform = \lambda a\!:\!\mathsf{real}.\ \lambda b\!:\!\mathsf{real}.\ \mathsf{prob}\ \ \begin{array}[t]{l}\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\\ a + x * (b - a)\end{array}\end{array}$$

We also encode a combination of a point-mass distribution and a uniform distribution over the same domain, which is neither a discrete distribution nor a continuous distribution:

$$\begin{array}{l}\mathsf{let}\ point\_uniform = \mathsf{prob}\ \ \begin{array}[t]{l}\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\\ \mathsf{if}\ x < 0.5\ \mathsf{then}\ 0.0\ \mathsf{else}\ x\end{array}\end{array}$$

Rich expressiveness

We now demonstrate the expressive power of PTP with a number of examples.

We encode a binomial distribution with parameters p and n_0 by exploiting probability terms:

$$\begin{array}{l}
\mathsf{let}\ binomial = \lambda p\!:\!\mathsf{real}.\ \lambda n_0\!:\!\mathsf{int}.\\
\quad \mathsf{let}\ bernoulli_p = bernoulli\ p\ \mathsf{in}\\
\quad \mathsf{let\ rec}\ binomial_p = \lambda n\!:\!\mathsf{int}.\\
\quad\quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{prob}\ 0\\
\quad\quad \mathsf{else}\ \mathsf{prob}\ \ \begin{array}[t]{l}\mathsf{sample}\ x\ \mathsf{from}\ binomial_p\ (n-1)\ \mathsf{in}\\ \mathsf{sample}\ b\ \mathsf{from}\ bernoulli_p\ \mathsf{in}\\ \mathsf{if}\ b\ \mathsf{then}\ 1 + x\ \mathsf{else}\ x\end{array}\\
\quad\quad \mathsf{in}\\
\quad binomial_p\ n_0
\end{array}$$

Here $binomial_p$ takes an integer n as input and returns a binomial distribution with parameters p and n.

If a probability distribution is defined in terms of a recursive process of generating samples, we can translate the definition into a recursive term. For example, we encode a geometric distribution with parameter p, which is a discrete distribution with infinite support, as follows:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{geometric\_rec} = \lambda p\!:\!\mathsf{real}. \\
\quad \mathsf{let}\ \mathit{bernoulli}_p = \mathit{bernoulli}\ p\ \mathsf{in} \\
\quad \mathsf{let\ rec}\ \mathit{geometric} = \mathsf{prob}\ \ \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{eif}\ b\ \mathsf{then}\ 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathit{geometric}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad 1 + x \\
\qquad \mathsf{in} \\
\quad \mathit{geometric}
\end{array}
$$

Here we use a recursive term $\mathit{geometric}$ of type $\bigcirc\mathsf{int}$. Equivalently we can use an expression fixed point construct:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{geometric\_efix} = \lambda p\!:\!\mathsf{real}.\ \ \mathsf{let}\ \mathit{bernoulli}_p = \mathit{bernoulli}\ p\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{prob}\ \ \mathsf{efix}\ \mathbf{geometric}\div\mathsf{int}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{eif}\ b\ \mathsf{then}\ 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{sample}\ x\ \mathsf{from\ prob}\ \mathbf{geometric}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 1 + x
\end{array}
$$

We encode an exponential distribution by using the inverse of its cumulative distribution function as a sampling function, which is known as the *inverse transform method*:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{exponential}_{1.0} = \mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathcal{S}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad -\mathsf{log}\ x
\end{array}
$$

The *rejection method*, which generates a sample from a probability distribution by repeatedly generating samples from other probability distributions until they satisfy a certain termination condition, can be implemented with a recursive term. For example, we encode a Gaussian distribution with mean m and variance σ^2 by the rejection method with respect to exponential distributions:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{bernoulli}_{0.5} = \mathit{bernoulli}\ 0.5 \\
\mathsf{let}\ \mathit{gaussian\_rejection} = \lambda m\!:\!\mathsf{real}.\ \lambda\sigma\!:\!\mathsf{real}. \\
\quad \mathsf{let\ rec}\ \mathit{gaussian} = \mathsf{prob}\ \ \mathsf{sample}\ y_1\ \mathsf{from}\ \mathit{exponential}_{1.0}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{sample}\ y_2\ \mathsf{from}\ \mathit{exponential}_{1.0}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{eif}\ y_2 \geq (y_1 - 1.0)^2/2.0\ \mathsf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_{0.5}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ m + \sigma * y_1\ \mathsf{else}\ m - \sigma * y_1 \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else\ unprob}\ \mathit{gaussian} \\
\qquad \mathsf{in} \\
\quad \mathit{gaussian}
\end{array}
$$

Since the probability p of $y_2 \geq (y_1 - 1.0)^2/2.0$ (the termination condition) is positive, the rejection method above terminates with probability $p + (1-p)p + (1-p)^2 p + \cdots = \frac{p}{1-(1-p)} = 1$. In this way, programmers can ensure that a particular sampling strategy by the rejection method terminates with probability 1.

We encode the joint distribution between two independent probability distributions using a product term. If M_P denotes $P(x)$ and M_Q denotes $Q(y)$, the following term denotes the joint distribution $\mathit{Prob}(x, y) \propto P(x)Q(y)$:

$$
\begin{array}{l}
\mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in} \\
\qquad\ \ \mathsf{sample}\ y\ \mathsf{from}\ M_Q\ \mathsf{in} \\
\qquad\ \ (x, y)
\end{array}
$$

For the joint distribution between two interdependent probability distributions, we use a conditional probability, which we represent as a lambda abstraction taking a regular value and returning a probability distribution. If M_P denotes $P(x)$ and M_Q denotes a conditional probability $Q(y|x)$, the following term denotes the joint distribution $Prob(x, y) \propto P(x)Q(y|x)$:

$$\begin{array}{ll}\mathsf{prob} & \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in} \\ & \mathsf{sample}\ y\ \mathsf{from}\ M_Q\ x\ \mathsf{in} \\ & (x, y)\end{array}$$

By returning y instead of (x, y), we compute the integration $Prob(y) = \int P(x)Q(y|x)dx$:

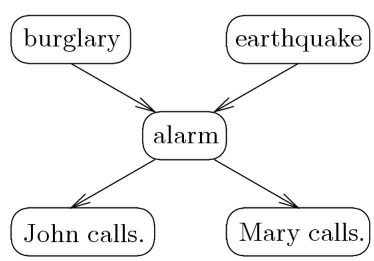
$$\begin{array}{ll}\mathsf{prob} & \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in} \\ & \mathsf{sample}\ y\ \mathsf{from}\ M_Q\ x\ \mathsf{in} \\ & y\end{array}$$

Due to lack of semantic constraints on sampling functions, we can specify probability distributions over unusual domains such as infinite data structures (*e.g.*, trees), function spaces, cyclic spaces (*e.g.*, angular values), and even probability distributions themselves. For example, we add two probability distributions over angular values in a straightforward way:

$$\begin{array}{ll}\mathsf{let}\ add\_angle = \lambda a_1 : \bigcirc\mathsf{real}.\ \lambda a_2 : \bigcirc\mathsf{real}.\ \mathsf{prob} & \mathsf{sample}\ s_1\ \mathsf{from}\ a_1\ \mathsf{in} \\ & \mathsf{sample}\ s_2\ \mathsf{from}\ a_2\ \mathsf{in} \\ & (s_1 + s_2)\ \mathsf{mod}\ (2.0 * \pi)\end{array}$$

With the modulo operation mod, we take into account the fact that an angle θ is identified with $\theta + 2\pi$.

As a simple application, we implement a belief network [66]:



We assume that $P_{alarm|burglary}$ denotes the probability distribution that the alarm goes off when a burglary happens; other variables of the form $P_{\cdot|\cdot}$ are interpreted in a similar way.

$$\begin{array}{l}\mathsf{let}\ alarm = \lambda(burglary, earthquake) : \mathsf{bool} \times \mathsf{bool}. \\ \quad \mathsf{if}\ burglary\ \mathsf{then}\ P_{alarm|burglary} \\ \quad \mathsf{else\ if}\ earthquake\ \mathsf{then}\ P_{alarm|\neg burglary \wedge earthquake} \\ \quad \mathsf{else}\ P_{alarm|\neg burglary \wedge \neg earthquake} \\ \mathsf{let}\ john\_calls = \lambda alarm : \mathsf{bool}. \\ \quad \mathsf{if}\ alarm\ \mathsf{then}\ P_{John\_calls|alarm} \\ \quad \mathsf{else}\ P_{John\_calls|\neg alarm} \\ \mathsf{let}\ mary\_calls = \lambda alarm : \mathsf{bool}. \\ \quad \mathsf{if}\ alarm\ \mathsf{then}\ P_{Mary\_calls|alarm} \\ \quad \mathsf{else}\ P_{Mary\_calls|\neg alarm}\end{array}$$

The conditional probabilities $alarm$, $john\_calls$, and $mary\_calls$ do not answer any query on the belief network and only describe its structure. In order to answer a specific query, we have to implement a corresponding probability distribution. For example, in order to answer "What is the probability $p_{Mary\_calls|John\_calls}$ that Mary calls when John calls?", we use $Q_{Mary\_calls|John\_calls}$ below, which essentially implements logic sampling [26]:

$$
\begin{array}{ll}
\mathsf{let\ rec}\ Q_{Mary\_calls|John\_calls} = \mathsf{prob} & \mathsf{sample}\ b\ \mathsf{from}\ P_{burglary}\ \mathsf{in} \\
& \mathsf{sample}\ e\ \mathsf{from}\ P_{earthquake}\ \mathsf{in} \\
& \mathsf{sample}\ a\ \mathsf{from}\ alarm\ (b, e)\ \mathsf{in} \\
& \mathsf{sample}\ j\ \mathsf{from}\ john\_calls\ a\ \mathsf{in} \\
& \mathsf{sample}\ m\ \mathsf{from}\ mary\_calls\ a\ \mathsf{in} \\
& \mathsf{eif}\ j\ \mathsf{then}\ m\ \mathsf{else\ unprob}\ Q_{Mary\_calls|John\_calls} \\
\quad \mathsf{in} & \\
Q_{Mary\_calls|John\_calls} &
\end{array}
$$

$P_{burglary}$ denotes the probability distribution that a burglary happens, and $P_{earthquake}$ the probability distribution that an earthquake happens. Then the mean of $Q_{Mary\_calls|John\_calls}$ gives $p_{Mary\_calls|John\_calls}$. We will see how to calculate $p_{Mary\_calls|John\_calls}$ in Section 3.4.

We can also implement most of the common operations on probability distributions. An exception is the Bayes operation $\sharp$ (which is used in the second update equation of the Bayes filter). $P \sharp Q$ results in a probability distribution R such that $R(x) = \eta P(x)Q(x)$ where η is a normalization constant ensuring $\int R(x)dx = 1.0$; if $P(x)Q(x)$ is zero for every x, then $P \sharp Q$ is undefined. Since it is difficult to achieve a general implementation of $P \sharp Q$, we usually make an additional assumption on P and Q to achieve a specialized implementation. For example, if we have a function p and a constant c such that $p(x) = kP(x) \leq c$ for a certain constant k, we can implement $P \sharp Q$ by the rejection method:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{let}\ bayes\_rejection = \lambda p{:}A \rightarrow \mathsf{real}.\ \lambda c{:}\mathsf{real}.\ \lambda Q{:}\bigcirc A.} \\
\quad \mathsf{let\ rec}\ bayes = \mathsf{prob} & \mathsf{sample}\ x\ \mathsf{from}\ Q\ \mathsf{in} \\
& \mathsf{sample}\ u\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in} \\
& \mathsf{eif}\ u < (p\ x)/c\ \mathsf{then}\ x\ \mathsf{else\ unprob}\ bayes \\
\qquad \mathsf{in} & \\
\quad bayes &
\end{array}
$$

We will see another implementation in Section 3.4.

High versatility

PTP allows high versatility in encoding probability distributions: given a probability distribution, we can exploit its unique properties and encode it in many different ways. For example, $exponential_{1.0}$ uses a logarithm function to encode an exponential distribution, but there is also an ingenious method (due to von

Neumann) that requires only addition and subtraction operations:

$$
\begin{array}{l}
\textsf{let } \mathit{exponential\_von\_Neumann}_{1.0} = \\
\quad \textsf{let rec } \mathit{search} = \lambda k \,{:}\, \textsf{real}.\, \lambda u \,{:}\, \textsf{real}.\, \lambda u_1 \,{:}\, \textsf{real}. \\
\quad\quad \textsf{prob} \;\; \textsf{sample } u' \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad\quad \textsf{eif } u < u' \textsf{ then } k + u_1 \\
\quad\quad\quad\quad \textsf{else} \\
\quad\quad\quad\quad\quad \textsf{sample } u \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad\quad\quad \textsf{eif } u \leq u' \textsf{ then unprob } (\mathit{search}\ k\ u\ u_1) \\
\quad\quad\quad\quad\quad \textsf{else} \\
\quad\quad\quad\quad\quad\quad \textsf{sample } u \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad\quad\quad\quad \textsf{unprob } (\mathit{search}\ (k + 1.0)\ u\ u) \\
\quad\quad \textsf{in} \\
\quad \textsf{prob} \;\; \textsf{sample } u \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad \textsf{unprob } (\mathit{search}\ 0.0\ u\ u)
\end{array}
$$

The recursive term in $\mathit{gaussian\_rejection}$ consumes at least three random numbers. We can encode a Gaussian distribution with only two random numbers:

$$
\begin{array}{l}
\textsf{let } \mathit{gaussian\_Box\_Muller} = \lambda m \,{:}\, \textsf{real}.\, \lambda \sigma \,{:}\, \textsf{real}. \\
\quad \textsf{prob} \;\; \textsf{sample } u \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad \textsf{sample } v \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad m + \sigma * \sqrt{-2.0 * \textsf{log } u} * \textsf{cos } (2.0 * \pi * v)
\end{array}
$$

We can also approximate a Gaussian distribution by exploiting the central limit theorem:

$$
\begin{array}{l}
\textsf{let } \mathit{gaussian\_central} = \lambda m \,{:}\, \textsf{real}.\, \lambda \sigma \,{:}\, \textsf{real}. \\
\quad \textsf{prob} \;\; \textsf{sample } x_1 \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad \textsf{sample } x_2 \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad\quad \cdots \\
\quad\quad\quad \textsf{sample } x_{12} \textsf{ from prob } \mathcal{S} \textsf{ in} \\
\quad\quad\quad m + \sigma * (x_1 + x_2 + \cdots + x_{12} - 6.0)
\end{array}
$$

The three examples above serve as evidence of high versatility of PTP: *the more we know about a probability distribution, the better we can encode it.*

All the examples in this section just rely on our intuition on sampling functions and do not actually prove the correctness of encodings. For example, we still do not know if $\mathit{bernoulli}$ indeed encodes a Bernoulli distribution, or equivalently, if the expression in it generates True with probability p. In the next section, we investigate how to formally prove the correctness of encodings.

3.3 Proving the correctness of encodings

When programming in PTP, we often ask *"What probability distribution characterizes outcomes of computing a given expression?"* The operational semantics of PTP does not directly answer this question because an expression computation returns only a single sample from a certain, yet unknown, probability distribution. Therefore we need a different methodology for interpreting expressions directly in terms of probability distributions.

We take a simple approach that appeals to our intuition on the meaning of expressions. We write $E \sim \mathit{Prob}$ if outcomes of computing E are distributed according to Prob. To determine Prob from E, we supply an infinite sequence of independent *random variables* from $U(0.0, 1.0]$ and analyze the result of computing E in terms of these random variables. If $E \sim Prob$, then E denotes a probabilistic computation for generating samples from $Prob$ and we regard $Prob$ as the denotation of prob E.

We illustrate the above approach with a few examples. In each example, R_i means the i-th random variable and R_i^∞ means the infinite sequence of random variables beginning from R_i (*i.e.*, $R_i R_{i+1} \cdots$). A random variable is regarded as a value because it represents real numbers in $(0.0, 1.0]$.

As a trivial example, consider prob $\mathcal{S}$. The computation of $\mathcal{S}$ proceeds as follows:

$$\mathcal{S} \ @ \ R_1^\infty \mapsto_{\mathsf{e}} R_1 \ @ \ R_2^\infty$$

Since the outcome is a random variable from $U(0.0, 1.0]$, we have $\mathcal{S} \sim U(0.0, 1.0]$.

As an example of discrete distribution, consider *bernoulli* p. The expression in it computes as follows:

$$\begin{array}{lll} & \mathsf{sample}\ x \ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\ x \le p & @\ R_1^\infty \\ \mapsto_{\mathsf{e}} & \mathsf{sample}\ x \ \mathsf{from}\ \mathsf{prob}\ R_1\ \mathsf{in}\ x \le p & @\ R_2^\infty \\ \mapsto_{\mathsf{e}} & R_1 \le p & @\ R_2^\infty \\ \mapsto_{\mathsf{e}} & \begin{array}{lll} \mathsf{True} & @\ R_2^\infty & \textit{if } R_1 \le p; \\ \mathsf{False} & @\ R_2^\infty & \textit{otherwise}. \end{array} & \end{array}$$

Since R_1 is a random variable from $U(0.0, 1.0]$, the probability of $R_1 \le p$ is p. Thus the outcome is True with probability p and False with probability $1.0 - p$, and *bernoulli* p denotes a Bernoulli distribution with parameter p.

As an example of continuous distribution, consider *uniform* $a\ b$. The expression in it computes as follows:

$$\begin{array}{lll} & \mathsf{sample}\ x \ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\ a + x * (b-a) & @\ R_1^\infty \\ \mapsto_{\mathsf{e}}^{*} & a + R_1 * (b-a) & @\ R_2^\infty \end{array}$$

Since we have

$$a + R_1 * (b-a) \in (a_0, b_0] \quad \textit{iff} \quad R_1 \in (\frac{a_0 - a}{b-a}, \frac{b_0 - a}{b-a}],$$

the probability that the outcome lies in $(a_0, b_0]$ is

$$\frac{b_0 - a}{b-a} - \frac{a_0 - a}{b-a} = \frac{b_0 - a_0}{b-a} \propto b_0 - a_0$$

where we assume $(a_0, b_0] \subset (a, b]$. Thus *uniform* $a\ b$ denotes a uniform distribution over $(a, b]$.

The following proposition shows that *binomial* $p\ n$ denotes a binomial distribution with parameters p and n, which we write as $Binomial_{p,n}$:

Proposition 3.4. *If* $binomial_p\ n \mapsto_{\mathsf{t}}^{*}$ prob $E_{p,n}$*, then* $E_{p,n} \sim Binomial_{p,n}$.

Proof. By induction on n.

Base case $n = 0$. We have $E_{p,n} = 0$. Since $Binomial_{p,n}$ is a point-mass distribution centered on 0, we have $E_{p,n} \sim Binomial_{p,n}$.

Inductive case $n > 0$. The computation of $E_{p,n}$ proceeds as follows:

$$\begin{array}{lll}
 & \mathsf{sample}\ x\ \mathsf{from}\ \mathit{binomial}_p\ (n-1)\ \mathsf{in} & \\
 & \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} & \\
 & \mathsf{if}\ b\ \mathsf{then}\ 1+x\ \mathsf{else}\ x & @\ R_1^\infty \\
\mapsto_{\mathsf{e}}^* & \mathsf{sample}\ x\ \mathsf{from\ prob}\ x_{p,n-1}\ \mathsf{in} & \\
 & \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} & \\
 & \mathsf{if}\ b\ \mathsf{then}\ 1+x\ \mathsf{else}\ x & @\ R_i^\infty \\
\mapsto_{\mathsf{e}}^* & \mathsf{sample}\ b\ \mathsf{from\ prob}\ b_p\ \mathsf{in} & \\
 & \mathsf{if}\ b\ \mathsf{then}\ 1+x_{p,n-1}\ \mathsf{else}\ x_{p,n-1} & @\ R_{i+1}^\infty \\
\mapsto_{\mathsf{e}}^* & \begin{array}{lll} 1+x_{p,n-1} & @\ R_{i+1}^\infty & \mathit{if}\ b_p = \mathsf{True}; \\ x_{p,n-1} & @\ R_{i+1}^\infty & \mathit{otherwise}. \end{array} &
\end{array}$$

By induction hypothesis, $\mathit{binomial}_p\ (n-1)$ generates a sample $x_{p,n-1}$ from $\mathit{Binomial}_{p,n-1}$ after consuming $R_1 \cdots R_{i-1}$ for some i (which is actually n). Since R_i is an independent random variable, $\mathit{bernoulli}_p$ generates a sample b_p that is independent of $x_{p,n-1}$. Then we obtain an outcome k with the probability of

$b_p = \mathsf{True}$ and $x_{p,n-1} = k-1$ or

$b_p = \mathsf{False}$ and $x_{p,n-1} = k$,

which is equal to $p * \mathit{Binomial}_{p,n-1}(k-1) + (1.0-p) * \mathit{Binomial}_{p,n-1}(k) = \mathit{Binomial}_{p,n}(k)$. Thus we have $E_{p,n} \sim \mathit{Binomial}_{p,n}$. □

As a final example, we show that $\mathit{geometric\_rec}\ p$ denotes a geometric distribution with parameter p. Suppose $\mathit{geometric} \mapsto_{\mathsf{t}}^* \mathsf{prob}\ E$ and $E \sim \mathit{Prob}$. The computation of E proceeds as follows:

$$\begin{array}{lll}
 & E & @\ R_1^\infty \\
\mapsto_{\mathsf{e}}^* & \mathsf{sample}\ b\ \mathsf{from\ prob}\ b_p\ \mathsf{in} & \\
 & \mathsf{eif}\ b\ \mathsf{then}\ 0 & \\
 & \mathsf{else}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathit{geometric}\ \mathsf{in} & @\ R_2^\infty \\
 & \qquad 1+x & \\
\mapsto_{\mathsf{e}}^* & \begin{array}{lll} 0 & @\ R_2^\infty & \mathit{if}\ b_p = \mathsf{True}; \\ \mathsf{sample}\ x\ \mathsf{from\ prob}\ E\ \mathsf{in}\ 1+x & @\ R_2^\infty & \mathit{otherwise}. \end{array} &
\end{array}$$

The first case happens with probability p and we get $\mathit{Prob}(0) = p$. In the second case, we compute the same expression E with R_2^∞. Since all random variables are independent, R_2^∞ can be thought of as a fresh sequence of random variables. Therefore the computation of E with R_2^∞ returns samples from the same probability distribution Prob and we get $\mathit{Prob}(1+k) = (1.0-p) * \mathit{Prob}(k)$. Solving the two equations, we get $\mathit{Prob}(k) = p * (1.0-p)^{k-1}$, which is the probability mass function for a geometric distribution with parameter p.

The above approach can be thought of as an adaption of the methodology established in simulation theory [10]. The proof of the correctness of a sampling method in simulation theory is easily transcribed into a proof similar to those shown in this section by interpreting random numbers in simulation theory as random variables in PTP. Thus PTP serves as a programming language in which sampling methods developed in simulation theory can be not only formally expressed but also formally reasoned about. All this is possible in part because an expression computation in PTP is provided with an infinite sequence of random numbers to consume, or equivalently, because of the use of generalized sampling functions as the mathematical basis.

An alternative approach would be to develop a denotational semantics based upon measure theory [65] by translating expressions into a measure-theoretic structure. Such a denotational semantics would be useful in answering such questions as:

- Does every expression in PTP result in a measurable sampling function? Or is it possible to write a pathological expression that corresponds to no measurable sampling function?

- Does every expression in PTP define a probability distribution? Or is it possible to write a pathological expression that defines no probability distribution?

If we ignore fixed point constructs of PTP, it is straightforward to translate expressions even directly into probability measures, since probability measures form a monad [22, 64] and expressions already follow a monadic syntax; a sampling expression $\mathcal{S}$ is translated into a Lebesgue measure over the unit interval $(0.0, 1.0]$. Let us write $[M]_{\mathsf{term}}$ for the denotation of term M. Then we can translate each expression E into a probability measure $[E]_{\mathsf{exp}}$ as follows:

- $[\mathsf{prob}\ E]_{\mathsf{term}} = [E]_{\mathsf{exp}}$.

- $[M]_{\mathsf{exp}}(S) = 1$ if $[M]_{\mathsf{term}}$ is in S.
  $[M]_{\mathsf{exp}}(S) = 0$ if $[M]_{\mathsf{term}}$ is not in S.

- $[\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E]_{\mathsf{exp}} = \int f d[M]_{\mathsf{term}}$ where a function f is defined as $f(x) = [E]_{\mathsf{exp}}$ and $\int f d[M]_{\mathsf{term}}$ is an integral of f over measure $[M]_{\mathsf{term}}$.

- $[\mathcal{S}]_{\mathsf{exp}}$ is a Lebesgue measure over the unit interval $(0.0, 1.0]$.

Note that the translation does not immediately reveal the probability measure corresponding to a given expression because it returns a *formula* for the probability measure rather than the probability measure itself. Hence, in order to obtain the probability measure, we have to go through essentially the same analysis as in the above approach. Ultimately we have to invert a sampling function represented by a given expression (because an event is assigned a probability proportional to the size of its inverse image under the sampling function), which may not be easy to do in a mechanical way in the presence of various operators.

Once we add fixed point constructs to PTP, expressions should be translated into a domain-theoretic structure because of recursive equations. Specifically a term $\mathsf{fix}\ x : \bigcirc A.\ M$ gives rise to a recursion equation on type $\bigcirc A$, and if a measure-theoretic structure is used for the denotation of terms of type $\bigcirc A$, it is difficult to solve the recursive equation; only with a domain-theoretic structure, the recursive equation can be given a theoretical treatment. The work by Jones [30] suggests that such a domain-theoretic structure could be constructed from a domain-theoretic model of real numbers [17], and we leave the development of a denotational semantics of PTP based upon domain theory as future work.

3.4 Approximate Computation in PTP

We have explored both how to encode probability distributions in PTP and how to interpret PTP in terms of probability distributions. In this section, we discuss another important aspect of probabilistic languages: reasoning about probability distributions.

The expressive power of a probabilistic language is an important factor affecting its practicality. Another important factor is its support for reasoning about probability distributions to determine their properties. In other words, it is important not only to be able to encode various probability distributions but also to be able to determine their properties such as means, variances, and probabilities of specific events. Unfortunately PTP does not support precise reasoning about probability distributions. That is, it does not permit a precise implementation of queries on probability distributions. Intuitively we must be able to calculate probabilities of specific events, but this is tantamount to inverting sampling functions. Hence, for example, we cannot calculate $p_{Mary\_calls|John\_calls}$ in the belief network example in Section 3.2 unless we analyze $Q_{Mary\_calls|John\_calls}$ to compute its mean in a similar way to the previous section.

Given that we cannot hope for precise reasoning in PTP, we choose to support approximate reasoning by the Monte Carlo method [40]. It approximately answers a query on a probability distribution by generating a large number of samples and then analyzing them. For example, we can approximate $p_{Mary\_calls|John\_calls}$, which is equal to the proportion of True's among an infinite number of samples from $Q_{Mary\_calls|John\_calls}$, by generating a large number of samples and counting the number of True's. Although the Monte Carlo method gives only an approximate answer, its accuracy improves with the number of samples. Moreover it is applicable to all kinds of probability distributions and is therefore particularly suitable for PTP.

In this section, we use the Monte Carlo method to implement the expectation query. We also show how to exploit the Monte Carlo method in implementing the Bayes operation. Both implementations are provided as primitive constructs of PTP.

3.4.1 Expectation query

Among common queries on probability distributions, the most important is the expectation query. The expectation of a function f with respect to a probability distribution P is the mean of f over P, which we write as $\int f dP$. Other queries may be derived as special cases of the expectation query. For example, the mean of a probability distribution over real numbers is the expectation of an identity function; the probability of an event $Event$ under a probability distribution P is $\int I_{Event} dP$ where $I_{Event}(x)$ is 1 if x is in $Event$ and 0 if not.

The Monte Carlo method states that we can approximate $\int f dP$ with a set of samples $V_1, \cdots, V_n$ from P:

$$\lim_{n \to \infty} \frac{f(V_1) + \cdots + f(V_n)}{n} = \int f dP$$

We introduce a term construct $\mathsf{expectation}$ which exploits the above equation:

$$\text{term} \quad M \quad ::= \quad \cdots \mid \mathsf{expectation}\ M_f\ M_P$$

$$\frac{\Gamma \vdash_{\mathsf{p}} M_f : A \to \mathsf{real} \quad \Gamma \vdash_{\mathsf{p}} M_P : \bigcirc A}{\Gamma \vdash_{\mathsf{p}} \mathsf{expectation}\ M_f\ M_P : \mathsf{real}}\ \mathsf{Exp}$$

$$\frac{\begin{array}{l} M_f \mapsto_{\mathsf{t}}^* f \quad M_P \mapsto_{\mathsf{t}}^* \mathsf{prob}\ E_P \\ \text{for } i = 1, \cdots, n \quad \text{new sampling sequence } \omega_i \quad E_P \ @\ \omega_i \mapsto_{\mathsf{e}}^* V_i \ @\ \omega_i' \quad f\ V_i \mapsto_{\mathsf{t}}^* v_i \end{array}}{\mathsf{expectation}\ M_f\ M_P \mapsto_{\mathsf{t}} \frac{\sum_i v_i}{n}}\ Exp$$

The rule Exp says that if M_f evaluates to a lambda abstraction denoting f and M_P evaluates to a probability term denoting P, then $\mathsf{expectation}\ M_f\ M_P$ reduces to an approximation of $\int f dP$. A run-time variable n (to be chosen by programmers) specifies the number of samples to generate from P. To evaluate $\mathsf{expectation}\ M_f\ M_P$, the run-time system initializes sampling sequence ω_i to generate sample V_i for $i = 1, \cdots, n$ (as indicated by new sampling sequence ω_i).

In the rule Exp, the accuracy of $\frac{\sum_i v_i}{n}$ is controlled not by PTP but solely by programmers. That is, PTP is not responsible for choosing a value of n (*e.g.*, by analyzing E_P) to guarantee a certain level of accuracy in estimating $\int f dP$. Rather it is programmers that decide a suitable value of n to achieve a desired level of accuracy (as well as an expression E_P for encoding P). Programmers are also allowed to pick up a particular value of n for each expectation query, rather than using the same value of n for all expectation queries. We do not consider this as a weakness of PTP, since E_P itself, chosen by programmers, affects the accuracy of $\frac{\sum_i v_i}{n}$ after all.

Although PTP provides no concrete guidance in choosing a value of n in the rule Exp, programmers can empirically determine a suitable value of n, namely the largest value of n that finishes an expectation query within a given time constraint. (A large value of n is better because it results in a more faithful approximation of P by samples V_i and a smaller difference between $\frac{\sum_i v_i}{n}$ and the true expectation $\int f dP$.) Ideally the time to evaluate expectation $M_f\ M_P$ should be directly proportional to n, but in practice, the computation of the same expression E_P may take a different time, especially if E_P expresses a recursive computation. Therefore programmers can try different values of n and find the largest one that finishes the expectation query within a given time constraint.

A problem with the above definition is that although expectation is a term construct, its reduction is probabilistic because of sampling sequence ω_i in the rule *Exp*. This violates the principle that a term evaluation is always deterministic, and now the same term may evaluate to different values if it contains expectation. In order not to violate the principle, we assume that sampling sequence ω_i in the rule *Exp* is uniquely determined by expression E_P.

Now we can calculate $p_{Mary\_calls|John\_calls}$ as:

$$\mathsf{expectation}\ (\lambda x\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ 1.0\ \mathsf{else}\ 0.0)\ Q_{Mary\_calls|John\_calls}$$

3.4.2 Bayes operation

The previous implementation of the Bayes operation $P \sharp Q$ assumes a function p and a constant c such that $p(x) = kP(x) \leq c$ for a certain constant k. It is, however, often difficult to find the optimal value of c (*i.e.*, the maximum value of $p(x)$) and we have to take a conservative estimate of c. The Monte Carlo method, in conjunction with importance sampling [40], allows us to dispense with c by approximating Q with a set of samples and $P \sharp Q$ with a set of weighted samples. We introduce a term construct bayes for the Bayes operation and an expression construct importance for importance sampling:

$$\begin{array}{llll} \text{term} & M & ::= & \cdots \mid \mathsf{bayes}\ M_p\ M_Q \\ \text{expression} & E & ::= & \cdots \mid \mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\} \end{array}$$

In the spirit of data abstraction, importance represents only an internal data structure and is not directly available to programmers.

$$\frac{\Gamma \vdash_{\mathsf{p}} M_p : A \rightarrow \mathsf{real} \quad \Gamma \vdash_{\mathsf{p}} M_Q : \bigcirc A}{\Gamma \vdash_{\mathsf{p}} \mathsf{bayes}\ M_p\ M_Q : \bigcirc A}\ \mathsf{Bayes}$$

$$\frac{\Gamma \vdash_{\mathsf{p}} V_i : A \quad \Gamma \vdash_{\mathsf{p}} w_i : \mathsf{real} \quad 1 \leq i \leq n}{\Gamma \vdash_{\mathsf{p}} \mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\} \div A}\ \mathsf{Imp}$$

$$\frac{\begin{array}{l} M_p \mapsto_{\mathsf{t}}^* p \quad M_Q \mapsto_{\mathsf{t}}^* \mathsf{prob}\ E_Q \\ \text{for } i = 1, \cdots, n \quad \text{new sampling sequence } \omega_i \quad E_Q \ @\ \omega_i \mapsto_{\mathsf{e}}^* V_i \ @\ \omega_i' \quad p\ V_i \mapsto_{\mathsf{t}}^* w_i \end{array}}{\mathsf{bayes}\ M_p\ M_Q \mapsto_{\mathsf{t}} \mathsf{prob}\ \mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\}}\ \mathit{Bayes}$$

$$\frac{\frac{\sum_{i=1}^{k-1} w_i}{S} < r \leq \frac{\sum_{i=1}^{k} w_i}{S} \quad \mathit{where} \quad S = \sum_{i=1}^{n} w_i}{\mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\} \ @\ r\omega \mapsto_{\mathsf{e}} V_k \ @\ \omega}\ \mathit{Imp}$$

The rule *Bayes* uses sampling sequences $\omega_1, \cdots, \omega_n$ initialized by the run-time system and approximates Q with n samples $V_1, \cdots, V_n$, where n is a run-time variable as in the rule *Exp*. Then it applies p to each sample V_i to calculates its weight w_i and creates a set $\{(V_i, w_i) | 1 \leq i \leq n\}$ of weighted samples as an argument to importance. The rule *Imp* implements importance sampling: we use a random number r to probabilistically select a sample V_k by taking into account the weights associated with all the samples. As with expectation, we decide to define bayes as a term construct with the assumption that sampling sequence ω_i in the rule *Bayes* is uniquely determined by expression E_Q.

3.4.3 expectation and bayes as expression constructs

Since their reduction involves sampling sequences, expectation and bayes could be defined as expression constructs so that the assumption on sampling sequence ω_i (in the rules Exp and $Bayes$) would be unnecessary. Still we choose to define expectation and bayes as term constructs for pragmatic reasons. Consider a probability distribution $P(s)$ defined in terms of probability distributions $Q(s)$ and $R(u)$:

$$P(s) = \eta Q(s) \int f(s,u) R(u) du$$

(A similar example is found in Section 5.3.) $P(s)$ is obtained by the Bayes operation between $Q(s)$ and $Prob(s) = \int f(s,u)R(u)du$, and is encoded in PTP as

$$\mathsf{bayes}\ (\lambda s\!:\!\_.\ \mathsf{expectation}\ (\lambda u\!:\!\_.\ M_f(s,u))\ M_Q)\ M_P$$

where M_P and M_Q are probability terms denoting P and Q, respectively, and M_f is a lambda abstraction denoting f. If expectation was an expression construct, however, it would be difficult to encode $P(s)$ because expression expectation $(\lambda u\!:\!\_.\ M_f(s,u))\ M_Q$ cannot be converted into a term. In essence, mathematically the expectation of a function with respect to a probability distribution and the result of a Bayes operation are always unique (if they exist), which in turn implies that if expectation and bayes are defined as expression constructs, we cannot write code involving expectations and Bayes operations in the same manner that we reason mathematically.

The actual implementation of PTP (to be presented in the next chapter) does not enforce the assumption on sampling sequence ω_i in the rules Exp and $Bayes$, which is unrealistic in practice and required only for the semantic clarity of PTP. Strictly speaking, therefore, term evaluations are not necessarily deterministic and there is no clear separation between terms and expressions in this regard. Since terms are not protected from computational effects (such as input/output and mutable references) and term evaluations do not always result in unique values anyway, non-deterministic term evaluations should not be regarded as a new problem. Thus expressions are best interpreted as a syntactic category dedicated to probabilistic computations only in the mathematical sense — strict adherence at the implementation level to the semantic distinction between terms and expressions (*e.g.*, defining expectation and bayes as expression constructs) would cost code readability without any apparent benefit.

3.4.4 Cost of generating random numbers

The essence of the Monte Carlo method is to trade accuracy for cost — it only gives approximate answers, but relieves programmers of the cost of exact computation (which can be even impossible in certain problems). Since PTP relies on the Monte Carlo method to reason about probability distributions, it is important for programmers to be able to determine the cost of the Monte Carlo method.

We decide to define the cost of the Monte Carlo method as proportional to the number of random numbers consumed. The decision is based upon the assumption that random number generation can account for a significant portion of the total computation time. (If the cost of random number generation was negligible, the number of random numbers consumed would be of little importance.) Under our implementation of PTP, random number generation for the following examples from Section 3.2 accounts for an average of 74.85% of the total computation time. The following table shows execution times (in seconds) and percentages of random number generation when generating 100,000 samples (on a Pentium III 500Mhz with 384 MBytes memory):

test case	execution time	random number generation (%)
$uniform$ 0.0 1.0	0.25	78.57
$binomial$ 0.25 16	4.65	64.84
$geometric\_efix$ 0.25	1.21	63.16
$gaussian\_rejection$ 2.5 5.0	1.13	77.78
$exponential\_von\_Neumann_{1.0}$	1.09	80.76
$gaussian\_Box\_Muller$ 2.0 4.0	0.57	77.27
$gaussian\_central$ 0.0 1.0	2.79	83.87
$Q_{Mary\_calls \mid John\_calls}$	21.35	72.57

In PTP, it is the programmers' responsibility to reason about the cost of generating random numbers, since for an expression computation judgment $E @ \omega \rightharpoonup V @ \omega'$, the length of the consumed sequence $\omega - \omega'$ is not observable. A analysis similar to those in Section 3.3 can be used to estimate the cost of obtaining a sample in terms of the number of random numbers consumed. In the case of $geometric\_rec\ p$, for example, the expected number n of random numbers consumed is calculated by solving the equation

$$n = 1 + (1 - p) * n$$

where 1 accounts for the random number generated from the Bernoulli distribution and $(1 - p)$ is the probability that another attempt is made to generate a sample from the same probability distribution. The same technique applies equally to the rejection method (*e.g.*, $gaussian\_rejection$).

3.5 Summary

Although conceptually simple, the idea of using sampling functions in specifying probability distributions is new in the history of probabilistic languages. PTP is an example of probabilistic language that indirectly expresses sampling functions in a monadic syntax. We could also choose a different syntax for expressing sampling functions. For example, the author [53] extends the lambda calculus with a *sampling construct* $\gamma.e$ to directly encodes sampling functions (γ is a formal argument and e denotes the body of a sampling function). The computation of $\gamma.e$ proceeds by generating a random number from $U(0.0, 1.0]$ and substituting it for γ in e. Compared with PTP, the resultant calculus facilitates the encoding of some probability distribution (*e.g.*, $\gamma.\gamma$ for $U(0.0, 1.0]$), but it also reduces code readability because every program fragment denotes a probability distribution and there is no separation between regular values and probabilistic computations.

The idea of using a monadic syntax for PTP was inspired by the stochastic lambda calculus of Ramsey and Pfeffer [64], whose denotational semantics is based upon the monad of probability measures, or the probability monad [22]. In implementing a query for generating samples from probability distributions, they note that the probability monad can also be interpreted in terms of sampling functions, both denotationally and operationally. In designing PTP, we take the opposite approach: first we use a monadic syntax for probabilistic computations and relate it directly to sampling functions; then we interpret it in terms of probability distributions.

The operational semantics of PTP can be presented in different styles. For example, expression computations could use a judgment of the form $E \stackrel{r_1 r_2 \cdots r_n}{\rightharpoonup} V$, meaning that expression E computes to sample V by consuming a finite sequence of random numbers $r_1, r_2, \cdots, r_n$. Although the new judgment better reflects the actual implementation of expression computation, we stick to the formulation given in this chapter to emphasize the logical foundation of PTP.

Chapter 4

Implementation

This chapter describes the implementation of PTP. Instead of implementing PTP as a complete programming language of its own, we choose to embed it in an existing functional language for two pragmatic reasons. First the conceptual basis of probabilistic computations in PTP is simple enough that it is easy to simulate all language constructs of PTP without any modification to the run-time system. Second we intend to use PTP for real applications in robotics, for which we wish to exploit advanced features such as a module system, an interface to foreign languages, and a graphics library. Hence building a complete compiler for PTP is not justified when extending an existing functional language is sufficient for examining the practicality of PTP.

We emphasize that embedding PTP in an existing functional language is different from adding a library to the host language. For example, the syntax of the host language is extended with the syntax of PTP, which is not the case when a library is added. Since the type system of PTP is also faithfully reflected in the host language, programmers can benefit from the type system of PTP even when programming in the host language environment. (A library can also partially reflect the type system of PTP through type abstraction, but not completely because of different syntax in the library.)

In our implementation, we use Objective CAML [2] as the host language. First we formulate a sound and complete translation of PTP in a simple call-by-value language which can be thought of a sublanguage of Objective CAML. Then we extend the syntax of Objective CAML using CAMLP4, a preprocessor for Objective CAML, to incorporate the syntax of PTP. The extended syntax is translated back in the original syntax.

4.1 Representation of sampling functions

Since a probability term denotes a probability distribution specified by a sampling function, the implementation of PTP translates probability terms into representations of sampling functions. We translate a probability term of type $\bigcirc A$ into a value of type A `prob`, where the type constructor `prob` is conceptually defined as follows:

$$\texttt{type}\ A\ \texttt{prob}\ =\ \texttt{real}^{\infty}\ \texttt{->}\ A * \texttt{real}^{\infty}$$

`real` is the type of real numbers, and we use $\texttt{real}^{\infty}$ for the type of infinite sequences of random numbers.

We simplify the definition of `prob` in two steps. First we implement real numbers of type `real` as floating point numbers of type `float` (as in Objective CAML). Second we dispense with infinite sequences of random numbers by using a global random number generator whenever fresh random numbers are needed to compute sampling expressions. Thus we use the following definition of `prob`:

$$\texttt{type}\ A\ \texttt{prob}\ =\ \texttt{unit}\ \texttt{->}\ A$$

type	A, B	$::=$	$A \rightarrow A \mid \bigcirc A \mid \mathsf{real}$
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\, M \mid M\ M \mid \mathsf{prob}\ E \mid r$
expression	E, F	$::=$	$M \mid \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \mid \mathcal{S} \mid \mathbf{x} \mid$ $\mathsf{efix}\ \mathbf{x} \div A.\, E$
value/sample	V	$::=$	$\lambda x\!:\!A.\, M \mid \mathsf{prob}\ E \mid r$
floating point number	r		
sampling sequence	ω	$::=$	$r_1 r_2 \cdots r_i \cdots \quad \textit{where } r_i \in (0.0, 1.0]$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \mathbf{x} \div A$

Figure 4.1: A fragment of PTP as the source language.

Here `unit` is the unit type which is inhabited only by a unit value $()$.

The use of type `float` instead of type `real` means that we use finite precision in representing sampling functions. Although the overhead of exact real arithmetic is not justified in those applications (*e.g.*, robotics) where we work with samples and approximations, programmers may demand higher precision than is supported by type `float`. As a contrived example, consider a binary distribution assigning probability 0.25 to True and probability 0.75 to False:

$$\begin{array}{ll}\mathsf{prob} & \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in} \\ & 2.0 * x \leq 0.5\end{array}$$

If type `float` uses only one bit in mantissa part (and $\mathcal{S}$ computes to either 0.5 or 1.0), the above probability term denotes a wrong probability distribution (namely a point-mass distribution centered on False); only with two or more bits in the mantissa part, it denotes the intended probability distribution. Therefore, while the finite precision supported by the implementation of PTP (64 bits floating point numbers in Objective CAML) is adequate for typical applications, it should also be noted that there can be sampling functions demanding higher precision and that errors induced by floating point numbers can be problematic in some applications.

We use the type constructor `prob` as an abstract datatype. That is, the definition of `prob` is not exposed to PTP and values of type A `prob` are accessed only via member functions. We provide two member functions: `prb` and `app`. `prb` builds a value of type A `prob` from a function of type `unit` $->$ A; it is actually defined as an identity function. `app` generates a sample from a value of type A `prob`; it applies its argument to a unit value. The interface and implementation of the abstract datatype `prob` are given as follows:

$\texttt{type}\ A\ \texttt{prob}$	$\texttt{type}\ A\ \texttt{prob}\ =\ \texttt{unit} -> A$
$\texttt{val prb}\ : (\texttt{unit} -> A) -> A\ \texttt{prob}$	$\texttt{let prb}\ = \texttt{fun}\ f\!:\!\texttt{unit} -> A.\ f$
$\texttt{val app}\ : A\ \texttt{prob} -> A$	$\texttt{let app}\ = \texttt{fun}\ f\!:\!A\ \texttt{prob}.\ f\ ()$

We use `prb` in translating probability terms and `app` in translating bind expressions. In conjunction with the use of the type constructor `prob` as an abstract data type, they provide a sound and complete translation of PTP, as shown in the next section.

4.2 Translation of PTP in a call-by-value language

We translate a fragment of PTP shown in Figure 4.1 in a call-by-value language shown in Figure 4.2. The source language excludes product types, which are straightforward to translate if the target language is extended with product types. We directly translate expression fixed point constructs without simulating

$$
\begin{array}{llll}
\text{type} & A, B & ::= & A \mathrel{\texttt{->}} A \mid A\,\texttt{prob} \mid \texttt{float} \mid \texttt{unit} \\
\text{expression} & e, f & ::= & x \mid \texttt{fun}\, x\!:\!A.\, e \mid e\; e \mid \texttt{prb}\, e \mid \texttt{app}\, e \mid r \mid \\
 & & & () \mid \texttt{random} \mid \texttt{fix}\, x\!:\!A.\, u \\
\text{value} & v & ::= & \texttt{fun}\, x\!:\!A.\, e \mid \texttt{prb}\, v \mid r \mid () \\
\text{function} & u & ::= & \texttt{fun}\, x\!:\!A.\, e \\
\text{floating point number} & r & & \\
\text{sampling sequence} & \omega & ::= & r_1 r_2 \cdots r_i \cdots \quad \textit{where } r_i \in (0.0, 1.0] \\
\text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 4.2: A call-by-value language as the target language.

$$
\frac{}{\Gamma, x : A \vdash_{\mathtt{v}} x : A}\ \texttt{Hyp} \qquad \frac{\Gamma, x : A \vdash_{\mathtt{v}} e : B}{\Gamma \vdash_{\mathtt{v}} \texttt{fun}\, x\!:\!A.\, e : A \mathrel{\texttt{->}} B}\ \texttt{Lam}
$$

$$
\frac{\Gamma \vdash_{\mathtt{v}} e_1 : A \mathrel{\texttt{->}} B \quad \Gamma \vdash_{\mathtt{v}} e_2 : A}{\Gamma \vdash_{\mathtt{v}} e_1\; e_2 : B}\ \texttt{App} \qquad \frac{\Gamma \vdash_{\mathtt{v}} e : \texttt{unit} \mathrel{\texttt{->}} A}{\Gamma \vdash_{\mathtt{v}} \texttt{prb}\, e : A\,\texttt{prob}}\ \texttt{Prb}
$$

$$
\frac{\Gamma \vdash_{\mathtt{v}} e : A\,\texttt{prob}}{\Gamma \vdash_{\mathtt{v}} \texttt{app}\, e : A}\ \texttt{Papp} \qquad \frac{}{\Gamma \vdash_{\mathtt{v}} r : \texttt{float}}\ \texttt{Float} \qquad \frac{}{\Gamma \vdash_{\mathtt{v}} () : \texttt{unit}}\ \texttt{Unit}
$$

$$
\frac{}{\Gamma \vdash_{\mathtt{v}} \texttt{random} : \texttt{float}}\ \texttt{Random} \qquad \frac{\Gamma, x : A \vdash_{\mathtt{v}} u : A}{\Gamma \vdash_{\mathtt{v}} \texttt{fix}\, x\!:\!A.\, u : A}\ \texttt{Fix}
$$

Figure 4.3: Typing rules of the target language.

them with fixed point constructs for terms. As the target language supports only floating point numbers, r in the source language is restricted to floating point numbers.

The target language is a call-by-value language extended with the abstract datatype `prob`. It has a single syntactic category consisting of expressions (because it does not distinguish between effect-free evaluations and effectful computations). As in PTP, every expression denotes a probabilistic computation and we say that an expression computes to a value. Note that fixed point constructs $\texttt{fix}\, x\!:\!A.\, u$ allow recursive expressions only over function types.

The type system of the target language is shown in Figure 4.3. It employs a typing judgment $\Gamma \vdash_{\mathtt{v}} e : A$, meaning that expression e has type A under typing context Γ. The rules `Prb` and `Papp` conform to the interface of the abstract datatype `prob`.

The operational semantics of the target language is shown in Figure 4.4. It employs an expression reduction judgment $e \ @\ \omega \mapsto_{\mathtt{v}} e' \ @\ \omega'$, meaning that the computation of e with sampling sequence ω reduces to the computation of e' with sampling sequence ω'. A capture-avoiding substitution $[e/x]f$ is defined in a standard way. The rule $\mathtt{E}_{\mathtt{AppPrb}}$ is defined according to the implementation of the abstract datatype `prob`. The rule $\mathtt{E}_{\mathtt{Random}}$ shows that `random`, like sampling expressions in PTP, consumes a random number in a given sampling sequence. We write $\mapsto_{\mathtt{v}}^*$ for the reflexive and transitive closure of $\mapsto_{\mathtt{v}}$.

Figure 4.5 shows the translation of the source language in the target language.[1] We overload the function $[\cdot]_{\mathtt{v}}$ for types, typing contexts, terms, and expressions. Both terms and expressions of type A in the source language are translated into expressions of type $[A]_{\mathtt{v}}$ in the target language. $[\mathsf{prob}\ E]_{\mathtt{v}}$ suspends the computation of $[E]_{\mathtt{v}}$ by building a function $\texttt{fun}\ \_\!:\!\texttt{unit}.\ [E]_{\mathtt{v}}$, just as $\mathsf{prob}\ E$ suspends the computation of E. Since the target language allows recursive expressions only over function types, an expression variable $\mathtt{x}$ of type A (*i.e.*, $\mathtt{x} \div A$) is translated into $x_{\mathtt{x}}\ ()$ where $x_{\mathtt{x}}$ is a special variable of type $\texttt{unit} \mathrel{\texttt{->}} [A]_{\mathtt{v}}$ annotated

[1] _ is a wildcard pattern for variables and types.

$$\frac{e \mathbin{@} \omega \mapsto_{\mathtt{v}} e' \mathbin{@} \omega'}{e\ f \mathbin{@} \omega \mapsto_{\mathtt{v}} e'\ f \mathbin{@} \omega'}\ \mathtt{E}_{\beta_{\mathtt{L}}} \qquad \frac{f \mathbin{@} \omega \mapsto_{\mathtt{v}} f' \mathbin{@} \omega'}{(\mathtt{fun}\ x\!:\!A.\ e)\ f \mathbin{@} \omega \mapsto_{\mathtt{v}} (\mathtt{fun}\ x\!:\!A.\ e)\ f' \mathbin{@} \omega'}\ \mathtt{E}_{\beta_{\mathtt{R}}}$$

$$\frac{}{(\mathtt{fun}\ x\!:\!A.\ e)\ v \mathbin{@} \omega \mapsto_{\mathtt{v}} [v/x]e \mathbin{@} \omega}\ \mathtt{E}_{\beta_{\mathtt{V}}} \qquad \frac{e \mathbin{@} \omega \mapsto_{\mathtt{v}} e' \mathbin{@} \omega'}{\mathtt{prb}\ e \mathbin{@} \omega \mapsto_{\mathtt{v}} \mathtt{prb}\ e' \mathbin{@} \omega'}\ \mathtt{E}_{\mathtt{Prb}}$$

$$\frac{e \mathbin{@} \omega \mapsto_{\mathtt{v}} e' \mathbin{@} \omega'}{\mathtt{app}\ e \mathbin{@} \omega \mapsto_{\mathtt{v}} \mathtt{app}\ e' \mathbin{@} \omega'}\ \mathtt{E}_{\mathtt{App}} \qquad \frac{}{\mathtt{app}\ \mathtt{prb}\ v \mathbin{@} \omega \mapsto_{\mathtt{v}} v\ () \mathbin{@} \omega}\ \mathtt{E}_{\mathtt{AppPrb}}$$

$$\frac{}{\mathtt{random} \mathbin{@} r\omega \mapsto_{\mathtt{v}} r \mathbin{@} \omega}\ \mathtt{E}_{\mathtt{Random}} \qquad \frac{}{\mathtt{fix}\ x\!:\!A.\ u \mathbin{@} \omega \mapsto_{\mathtt{v}} [\mathtt{fix}\ x\!:\!A.\ u/x]u \mathbin{@} \omega}\ \mathtt{E}_{\mathtt{Fix}}$$

Figure 4.4: Operational semantics of the target language.

$$\begin{array}{rcl}
[A \to B]_{\mathtt{v}} & = & [A]_{\mathtt{v}}\ \texttt{->}\ [B]_{\mathtt{v}} \\
[\bigcirc A]_{\mathtt{v}} & = & [A]_{\mathtt{v}}\ \mathtt{prob} \\
[\mathsf{real}]_{\mathtt{v}} & = & \mathtt{float} \\
\\
[\cdot]_{\mathtt{v}} & = & \cdot \\
[\Gamma, x : A]_{\mathtt{v}} & = & [\Gamma]_{\mathtt{v}}, x : [A]_{\mathtt{v}} \\
[\Gamma, \mathbf{x} \div A]_{\mathtt{v}} & = & [\Gamma]_{\mathtt{v}}, x_{\mathbf{x}} : \mathtt{unit}\ \texttt{->}\ [A]_{\mathtt{v}} \\
\\
[x]_{\mathtt{v}} & = & x \\
[\lambda x\!:\!A.\ M]_{\mathtt{v}} & = & \mathtt{fun}\ x\!:\![A]_{\mathtt{v}}.\ [M]_{\mathtt{v}} \\
[M\ N]_{\mathtt{v}} & = & [M]_{\mathtt{v}}\ [N]_{\mathtt{v}} \\
[\mathsf{prob}\ E]_{\mathtt{v}} & = & \mathtt{prb}\ (\mathtt{fun}\ _\!:\!\mathtt{unit}.\ [E]_{\mathtt{v}}) \\
[r]_{\mathtt{v}} & = & r \\
[\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E]_{\mathtt{v}} & = & (\mathtt{fun}\ x\!:\!_.\ [E]_{\mathtt{v}})\ (\mathtt{app}\ [M]_{\mathtt{v}}) \\
[\mathcal{S}]_{\mathtt{v}} & = & \mathtt{random} \\
[\mathbf{x}]_{\mathtt{v}} & = & x_{\mathbf{x}}\ () \\
[\mathsf{efix}\ \mathbf{x}\!\div\! A.\ E]_{\mathtt{v}} & = & (\mathtt{fix}\ x_{\mathbf{x}}\!:\!\mathtt{unit}\ \texttt{->}\ [A]_{\mathtt{v}}.\ \mathtt{fun}\ _\!:\!\mathtt{unit}.\ [E]_{\mathtt{v}})\ ()
\end{array}$$

Figure 4.5: Translation of the source language.

with $\mathbf{x}$; if the target language allowed recursive expressions over any type, $\mathbf{x}$ and $\mathsf{efix}\ \mathbf{x}\!\div\! A.\ E$ could be translated into $x_{\mathbf{x}}$ and $\mathtt{fix}\ x_{\mathbf{x}}\!:\![A]_{\mathtt{v}}.\ [E]_{\mathtt{v}}$, respectively.[2]

Propositions 4.1 and 4.2 show that the translation is faithful to the type system of the source language. Proposition 4.1 proves the soundness of the translation: a well-typed term or expression in the source language is translated into a well-typed expression in the target language. Proposition 4.2 proves the completeness of the translation: only a well-typed term or expression in the source language is translated into a well-type expression in the target language.

Proposition 4.1.
If $\Gamma \vdash_{\mathsf{p}} M : A$, *then* $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [M]_{\mathtt{v}} : [A]_{\mathtt{v}}$.
If $\Gamma \vdash_{\mathsf{p}} E \div A$, *then* $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [E]_{\mathtt{v}} : [A]_{\mathtt{v}}$.

Proof. By simultaneous induction on the structure of M and E. □

Proposition 4.2.

[2] In the Objective CAML syntax, $[\mathsf{efix}\ \mathbf{x}\!\div\! A.\ E]_{\mathtt{v}}$ can be rewritten as `let rec` $x_{\mathbf{x}}$ `() =` $[E]_{\mathtt{v}}$ `in` $x_{\mathbf{x}}$ `()`.

If $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [M]_{\mathtt{v}} : A$*, then there exists* B *such that* $A = [B]_{\mathtt{v}}$ *and* $\Gamma \vdash_{\mathtt{p}} M : B$*.*
If $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [E]_{\mathtt{v}} : A$*, then there exists* B *such that* $A = [B]_{\mathtt{v}}$ *and* $\Gamma \vdash_{\mathtt{p}} E \div B$*.*

Proof. By simultaneous induction on the structure of M and E. The conclusion in the first clause also implies $\Gamma \vdash_{\mathtt{p}} M \div B$. An interesting case is when $E = \mathbf{x}$.
Case $E = \mathbf{x}$:
$[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [\mathbf{x}]_{\mathtt{v}} : A$ — by assumption
$[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} x_{\mathbf{x}}\ () : A$ — because $[\mathbf{x}]_{\mathtt{v}} = x_{\mathbf{x}}\ ()$
$x_{\mathbf{x}} : \mathtt{unit} \mathbin{\texttt{->}} A \in [\Gamma]_{\mathtt{v}}$ — by `App` and `Unit`
Since $x_{\mathbf{x}}$ is a special variable annotated with expression variable $\mathbf{x}$,
$x_{\mathbf{x}} \div B \in \Gamma$ and $A = [B]_{\mathtt{v}}$ for some B.
$A = [B]_{\mathtt{v}}$ and $\Gamma \vdash_{\mathtt{p}} E \div B$. □

The translation is also faithful to the operational semantics of the source language. We first show that the translation is sound: a term reduction in the source language is translated into a corresponding expression reduction which consumes no random number (Proposition 4.6); an expression reduction in the source language is translated into a corresponding sequence of expression reductions which consumes the same sequence of random numbers (Proposition 4.7). Note that in Proposition 4.7, $[E]_{\mathtt{v}}$ does not directly reduce to $[F]_{\mathtt{v}}$; instead it reduces to an expression e to which $[F]_{\mathtt{v}}$ eventually reduces without consuming random numbers.

Lemma 4.3. $[[M/x]N]_{\mathtt{v}} = [[M]_{\mathtt{v}}/x][N]_{\mathtt{v}}$ *and* $[[M/x]E]_{\mathtt{v}} = [[M]_{\mathtt{v}}/x][E]_{\mathtt{v}}$*.*

Proof. By simultaneous induction on the structure of N and E. □

Lemma 4.4.
$[[\mathsf{efix}\ \mathbf{x}\div A.\ E/\mathbf{x}]M]_{\mathtt{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}\mathbin{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathtt{v}}.\ \mathtt{fun}\ \_\mathbin{:}\mathtt{unit}.\ [E]_{\mathtt{v}})/x_{\mathbf{x}}][M]_{\mathtt{v}}$.
$[[\mathsf{efix}\ \mathbf{x}\div A.\ E/\mathbf{x}]F]_{\mathtt{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}\mathbin{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathtt{v}}.\ \mathtt{fun}\ \_\mathbin{:}\mathtt{unit}.\ [E]_{\mathtt{v}})/x_{\mathbf{x}}][F]_{\mathtt{v}}$.

Proof. By simultaneous induction on the structure of M and F. □

Corollary 4.5.
$[[\mathsf{efix}\ \mathbf{x}\div A.\ E/\mathbf{x}]E]_{\mathtt{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}\mathbin{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathtt{v}}.\ \mathtt{fun}\ \_\mathbin{:}\mathtt{unit}.\ [E]_{\mathtt{v}})/x_{\mathbf{x}}][E]_{\mathtt{v}}$.

Proposition 4.6.
If $M \mapsto_{\mathtt{t}} N$*, then* $[M]_{\mathtt{v}}\ @\ \omega \mapsto_{\mathtt{v}} [N]_{\mathtt{v}}\ @\ \omega$ *for any sampling sequence* ω*.*

Proof. By induction on the structure of the derivation of $M \mapsto_{\mathtt{t}} N$.
Case $\dfrac{M \mapsto_{\mathtt{t}} M'}{M\ N \mapsto_{\mathtt{t}} M'\ N}\ T_{\beta_L}$:
$[M]_{\mathtt{v}}\ @\ \omega \mapsto_{\mathtt{v}} [M']_{\mathtt{v}}\ @\ \omega$ — by induction hypothesis
$[M\ N]_{\mathtt{v}} = [M]_{\mathtt{v}}\ [N]_{\mathtt{v}}$
$[M]_{\mathtt{v}}\ [N]_{\mathtt{v}}\ @\ \omega \mapsto_{\mathtt{v}} [M']_{\mathtt{v}}\ [N]_{\mathtt{v}}\ @\ \omega$ — by $\mathtt{E}_{\beta_{\mathtt{L}}}$
$[M']_{\mathtt{v}}\ [N]_{\mathtt{v}} = [M'\ N]_{\mathtt{v}}$
Case $\dfrac{N \mapsto_{\mathtt{t}} N'}{(\lambda x\mathbin{:}A.\ M)\ N \mapsto_{\mathtt{t}} (\lambda x\mathbin{:}A.\ M)\ N'}\ T_{\beta_R}$:
$[N]_{\mathtt{v}}\ @\ \omega \mapsto_{\mathtt{v}} [N']_{\mathtt{v}}\ @\ \omega$ — by induction hypothesis
$[(\lambda x\mathbin{:}A.\ M)\ N]_{\mathtt{v}} = (\mathtt{fun}\ x\mathbin{:}[A]_{\mathtt{v}}.\ [M]_{\mathtt{v}})\ [N]_{\mathtt{v}}$
$(\mathtt{fun}\ x\mathbin{:}[A]_{\mathtt{v}}.\ [M]_{\mathtt{v}})\ [N]_{\mathtt{v}}\ @\ \omega \mapsto_{\mathtt{v}} (\mathtt{fun}\ x\mathbin{:}[A]_{\mathtt{v}}.\ [M]_{\mathtt{v}})\ [N']_{\mathtt{v}}\ @\ \omega$ — by $\mathtt{E}_{\beta_{\mathtt{R}}}$
$(\mathtt{fun}\ x\mathbin{:}[A]_{\mathtt{v}}.\ [M]_{\mathtt{v}})\ [N']_{\mathtt{v}} = [(\lambda x\mathbin{:}A.\ M)\ N']_{\mathtt{v}}$
Case $\dfrac{}{(\lambda x\mathbin{:}A.\ M)\ V \mapsto_{\mathtt{t}} [V/x]M}\ T_{\beta_V}$:

$[(\lambda x\!:\!A.\,M)\;V]_{\mathtt{v}} = (\mathtt{fun}\;x\!:\![A]_{\mathtt{v}}.\,[M]_{\mathtt{v}})\;[V]_{\mathtt{v}}$
$(\mathtt{fun}\;x\!:\![A]_{\mathtt{v}}.\,[M]_{\mathtt{v}})\;[V]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}} [[V]_{\mathtt{v}}/x][M]_{\mathtt{v}} \;@\; \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$[[V]_{\mathtt{v}}/x][M]_{\mathtt{v}} = [[V/x]M]_{\mathtt{v}}$ by Lemma 4.3
□

Proposition 4.7.
If $E \;@\; \omega \mapsto_{\mathsf{e}} F \;@\; \omega'$, there exists e such that $[E]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$ and $[F]_{\mathtt{v}} \;@\; \omega' \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$.

Proof. By induction on the structure of the derivation of $E \;@\; \omega \mapsto_{\mathsf{e}} F \;@\; \omega'$. We consider two interesting cases.

Case $\dfrac{E \;@\; \omega \mapsto_{\mathsf{e}} E' \;@\; \omega'}{\mathsf{sample}\;x\;\mathsf{from\;prob}\;E\;\mathsf{in}\;F \;@\; \omega \mapsto_{\mathsf{e}} \mathsf{sample}\;x\;\mathsf{from\;prob}\;E'\;\mathsf{in}\;F \;@\; \omega'}\;E_{BindR}$:

$[E]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$ where $[E']_{\mathtt{v}} \;@\; \omega' \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$ by induction hypothesis
$[\mathsf{sample}\;x\;\mathsf{from\;prob}\;E\;\mathsf{in}\;F]_{\mathtt{v}} = (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;(\mathtt{app}\;(\mathtt{prb}\;(\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}})))$
$(\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;(\mathtt{app}\;(\mathtt{prb}\;(\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}}))) \;@\; \omega$
$\mapsto_{\mathtt{v}} (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;((\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}})\;()) \;@\; \omega$ by $\mathtt{E}_{\mathtt{AppPrb}}$
$\mapsto_{\mathtt{v}} (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;[E]_{\mathtt{v}} \;@\; \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$\mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;e \;@\; \omega'$ by $[E]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$
$[\mathsf{sample}\;x\;\mathsf{from\;prob}\;E'\;\mathsf{in}\;F]_{\mathtt{v}} = (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;(\mathtt{app}\;(\mathtt{prb}\;(\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E']_{\mathtt{v}})))$
$(\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;(\mathtt{app}\;(\mathtt{prb}\;(\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E']_{\mathtt{v}}))) \;@\; \omega'$
$\mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;[E']_{\mathtt{v}} \;@\; \omega'$ by $\mathtt{E}_{\mathtt{AppPrb}}$ and $\mathtt{E}_{\beta_{\mathtt{V}}}$
$\mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\;x\!:\!\_.\,[F]_{\mathtt{v}})\;e \;@\; \omega'$ by $[E']_{\mathtt{v}} \;@\; \omega' \mapsto_{\mathtt{v}}^{*} e \;@\; \omega'$

Case $\dfrac{}{\mathsf{efix}\;\mathbf{x}\!\div\!A.\,E \;@\; \omega \mapsto_{\mathsf{e}} [\mathsf{efix}\;\mathbf{x}\!\div\!A.\,E/\mathbf{x}]E \;@\; \omega}\;\mathit{Efix}$:

$[\mathsf{efix}\;\mathbf{x}\!\div\!A.\,E]_{\mathtt{v}} = (\mathtt{fix}\;x_{\mathbf{x}}\!:\!\mathtt{unit} \mathbin{-\!\!>} [A]_{\mathtt{v}}.\,\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}})\;()$
$(\mathtt{fix}\;x_{\mathbf{x}}\!:\!\mathtt{unit} \mathbin{-\!\!>} [A]_{\mathtt{v}}.\,\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}})\;() \;@\; \omega$
$\mapsto_{\mathtt{v}} (\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[\mathtt{fix}\;x_{\mathbf{x}}\!:\!\mathtt{unit} \mathbin{-\!\!>} [A]_{\mathtt{v}}.\,\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}}/x_{\mathbf{x}}][E]_{\mathtt{v}})\;() \;@\; \omega$ by $\mathtt{E}_{\mathtt{Fix}}$
$\mapsto_{\mathtt{v}}^{*} [\mathtt{fix}\;x_{\mathbf{x}}\!:\!\mathtt{unit} \mathbin{-\!\!>} [A]_{\mathtt{v}}.\,\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}}/x_{\mathbf{x}}][E]_{\mathtt{v}} \;@\; \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$[[\mathsf{efix}\;\mathbf{x}\!\div\!A.\,E/\mathbf{x}]E]_{\mathtt{v}} = [\mathtt{fix}\;x_{\mathbf{x}}\!:\!\mathtt{unit} \mathbin{-\!\!>} [A]_{\mathtt{v}}.\,\mathtt{fun}\;\_\!:\!\mathtt{unit}.\,[E]_{\mathtt{v}}/x_{\mathbf{x}}][E]_{\mathtt{v}}$ by Corollary 4.5
□

The completeness of the translation states that only a valid term or expression reduction in the source language is translated into a corresponding sequence of expression reductions in the target language. In other words, a term or expression that cannot be further reduced in the source language is translated into an expression whose reduction eventually gets stuck. To simplify the presentation, we introduce three judgments, all of which express that a term or expression does not further reduces.

- $M \mapsto_{\mathtt{t}} \bullet$ means that there exists no term to which M reduces.
- $E \;@\; \omega \mapsto_{\mathsf{e}} \bullet$ means that there exists no expression to which E reduces.
- $e \;@\; \omega \mapsto_{\mathtt{v}} \bullet$ means that there exists no expression to which e reduces (in the target language).

Corollary 4.9 proves the completeness of the translation for terms; Proposition 4.10 proves the completeness of the translation for expressions.

Proposition 4.8. *If $[M]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}} e \;@\; \omega'$, then $e = [N]_{\mathtt{v}}$, $\omega = \omega'$, and $M \mapsto_{\mathtt{t}} N$.*

Proof. By induction on the structure of M. We only need to consider the case $M = M_1\;M_2$. There are three cases of the structure of $[M_1\;M_2]_{\mathtt{v}} \;@\; \omega \mapsto_{\mathtt{v}} e \;@\; \omega'$ (corresponding to the rules $\mathtt{E}_{\beta_{\mathtt{L}}}$, $\mathtt{E}_{\beta_{\mathtt{R}}}$, and $\mathtt{E}_{\beta_{\mathtt{V}}}$). The case for the rule $\mathtt{E}_{\beta_{\mathtt{V}}}$ uses Lemma 4.3. □

Corollary 4.9. *If $M \mapsto_{\mathtt{t}} \bullet$, then $[M]_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet$ for any sampling sequence ω.*

Proposition 4.10. *If $E \mathbin{@} \omega \mapsto_{\mathtt{e}} \bullet$, then there exists e such that $[E]_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}}^{*} e \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet$.*

Proof. By induction on the structure of E. We consider two cases $E = M$ and $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F$; the remaining cases are all trivial.

Case $E = M$, $[E]_{\mathtt{v}} = [M]_{\mathtt{v}}$:

- $M \mapsto_{\mathtt{t}} \bullet$ — by E_{Term}
- $[M]_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet$ — by Corollary 4.9
- We let $e = [M]_{\mathtt{v}}$.

Case $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F$, $[E]_{\mathtt{v}} = (\mathtt{fun}\ x\,{:}\,\_.\ [F]_{\mathtt{v}})\ \mathtt{app}\ [M]_{\mathtt{v}}$:

If $M \neq \mathsf{prob}\ \cdot$,

- $M \mapsto_{\mathtt{t}} \bullet$ — by E_{Bind}
- $[M]_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet$ — by Corollary 4.9
- The rule $\mathtt{E}_{\mathtt{App}}$ does not apply to $[E]_{\mathtt{v}}$.
- The rule $\mathtt{E}_{\mathtt{AppPrb}}$ does not apply to $[E]_{\mathtt{v}}$. — $[M]_{\mathtt{v}} \neq \mathtt{prb}\ \cdot$
- We let $e = [E]_{\mathtt{v}}$.

If $M = \mathsf{prob}\ E'$, $E' \neq V$,

- $E' \mathbin{@} \omega \mapsto_{\mathtt{e}} \bullet$ — by E_{BindR}
- There exists e' such that $[E']_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}}^{*} e' \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet$ by induction hypothesis.

$$
\begin{array}{lr}
[E]_{\mathtt{v}} \mathbin{@} \omega & \\
\quad \mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\ x\,{:}\,\_.\ [F]_{\mathtt{v}})\ [E']_{\mathtt{v}} \mathbin{@} \omega & [M]_{\mathtt{v}} = \mathtt{prb}\ \mathtt{fun}\ \_{:}\mathtt{unit}.\ [E']_{\mathtt{v}} \\
\quad \mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\ x\,{:}\,\_.\ [F]_{\mathtt{v}})\ e' \mathbin{@} \omega & [E']_{\mathtt{v}} \mathbin{@} \omega \mapsto_{\mathtt{v}}^{*} e' \mathbin{@} \omega \\
\quad \mapsto_{\mathtt{v}} \bullet & e' \mathbin{@} \omega \mapsto_{\mathtt{v}} \bullet
\end{array}
$$

We let $e = (\mathtt{fun}\ x\,{:}\,\_.\ [F]_{\mathtt{v}})\ e'$.

If $M = \mathsf{prob}\ V$, then $E \mathbin{@} \omega \mapsto_{\mathtt{e}} \bullet$ does not hold because of the rule E_{BindV}. $\square$

The target language can be thought of as a sublanguage of Objective CAML in which the abstract datatype `prob` is built-in and `random` is implemented as `Random.float 1.0`.[3] Since Objective CAML also serves as the host language for PTP, we need to extend the syntax of Objective CAML to incorporate the syntax of PTP. The extended syntax is then translated back in the original syntax of Objective CAML using the function $[\cdot]_{\mathtt{v}}$. The next section gives the definition of the extended syntax.

4.3 Extended syntax

We use CAMLP4 to conservatively extend the syntax of Objective CAML, which is assumed to be specified by a non-terminal $\langle term \rangle$ (corresponding to terms in PTP), with a new non-terminal $\langle expr \rangle$ (corresponding to expressions in PTP); $\langle patt \rangle$ is a non-terminal for patterns and $\langle id \rangle$ for identifiers:

$\langle term \rangle$	::=	$\cdots$ | `PROB { `$\langle expr \rangle$` }`	probability term
$\langle expr \rangle$	::=	`[`$\langle term \rangle$`]` |	term as an expr.
		`sample `$\langle patt \rangle$` from `$\langle term \rangle$` in `$\langle expr \rangle$ |	bind expr.
		`UNIFORM` |	sampling expr.
		`efix `$\langle id \rangle$` -> `$\langle expr \rangle$ |	expr. fixed.p.c.
		`#`$\langle id \rangle$ |	expr. variable
		`unprob `$\langle term \rangle$ |	unprob
		`eif `$\langle term \rangle$` then `$\langle expr \rangle$` else `$\langle expr \rangle$	eif

[3] To be strict, `random` would be implemented as `1.0 -. Random.float 1.0`.

`[`$\langle term \rangle$`]` explicitly marks a term as an instance of expression. `#`$\langle id \rangle$ refers to an expression variable $\langle id \rangle$. All other expression constructs resemble their counterparts in Chapter 3.

As an example, we encode a Bernoulli distribution over type `bool` as follows:

```
let bernoulli = fun p ->
  PROB { sample x from PROB { UNIFORM } in
         [if x <= p then true else false] }
```

A geometric distribution is encoded with an expression fixed point construct as follows:

```
let geometric = fun p ->
  let bernoulli_p = bernoulli p in
  PROB {
    efix geo ->
      sample b from bernoulli_p in
      eif b then [0]
      else
        sample x from PROB { #geo } in
        [1 + x]
  }
```

All other examples in Section 3.2 can be encoded in a similar way.

4.4 Approximate computation

In PTP, reasoning about a probability distribution is accomplished by generating multiple samples and analyzing them. The implementation of PTP provides two functions for generating independent samples from a given probability distribution:

```
type 'a set
type 'a wset
val prob_to_set : 'a prob -> 'a set
val prob_to_wset : 'a prob -> ('a -> float) -> 'a wset
```

- `'a set` is a datatype for sets of samples of type `'a`.
- `'a wset` is a datatype for sets of weighted samples of type `'a`. Each sample is assigned a weight of type `float` and `'a wset` may be thought of as `('a * float) set`. All weights are normalized (*i.e.*, their sum is 1.0).
- `prob_to_set p` generates samples from `p` by evaluating app p repeatedly.
- `prob_to_wset p f` generates samples from `p` and assigns to each sample V a weight of `f` V.

Programmers can specify the number of samples generated from `prob_to_set` and `prob_to_wset`, thereby controlling the accuracy in approximating probability distributions.

The implementation of PTP provides two functions for applying the Monte Carlo method:

```
val set_monte_carlo : 'a set -> ('a -> float) -> float
val wset_monte_carlo : 'a wset -> ('a -> float) -> float
```

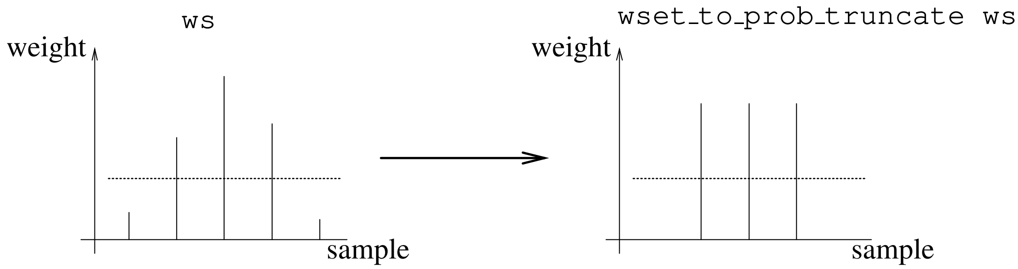


Figure 4.6: `wset_to_prob_truncate`.

- `set_monte_carlo s f` returns $\frac{\sum_{V \in \mathtt{s}} \mathtt{f}\ V}{|\mathtt{s}|}$.
- `wset_monte_carlo ws f` returns $\sum_{(V,w) \in \mathtt{ws}} (\mathtt{f}\ V) \cdot w$.

The following two functions convert sets and weighted sets back to probability distributions:

```
val set_to_prob_resample : 'a set -> 'a prob
val wset_to_prob_resample : 'a wset -> 'a prob
```

- `set_to_prob_resample s` returns a uniform distribution over `s`.
- `wset_to_prob_resample ws` returns prob importance `ws` which performs importance sampling on `ws` to select samples.

Now the expectation query (in Section 3.4.1) and the Bayes operation (in Section 3.4.2) are implemented by composing these functions:

expectation f p = `set_monte_carlo (prob_to_set p) f`

bayes f p = `wset_to_prob_resample (prob_to_wset p f)`

The implementation of PTP also provides a function for approximating the support of a given probability distribution. Since the support of an arbitrary probability distribution cannot be calculated accurately, we represent it as a uniform distribution:

```
val wset_to_prob_truncate : 'a wset -> 'a prob
```

`wset_to_prob_truncate ws` returns a uniform distribution over n samples of highest weights in `ws`, where n is the parameter specifying the number of samples generated by `prob_to_set` and `prob_to_wset`. Figure 4.6 illustrates how `wset_to_prob_truncate` works. `ws` has five samples in it, and `wset_to_prob_truncate` is invoked when the parameter n is set to three. The two samples with lowest weights perish, and all the surviving samples are assigned the same weight.

`wset_to_prob_truncate` is useful particularly when we want to extract a small number of samples of high weights from a probability distribution. For (an approximation of) the uniform distribution over the support of `p`, we use `wset_to_prob_truncate (prob_to_wset p (fun _ -> 1.0))`, where `(fun _ -> 1.0)` is a constant Objective CAML function returning `1.0`.

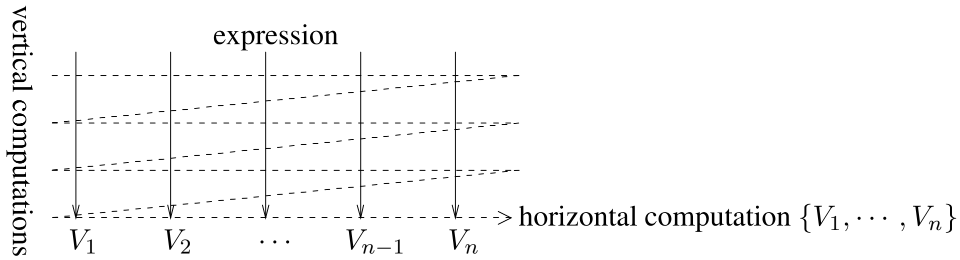


Figure 4.7: Horizontal and vertical computations.

4.5 Simultaneous computation of multiple samples

The implementation of PTP uses a simple strategy to generate multiple samples from a given probability distribution: compute the same expression repeatedly. An alternative strategy is to perform a single parallel computation that simulates multiple independent computations. To distinguish the two kinds of computations, we refer to the former strategy as *vertical computations* and the latter strategy as a *horizontal computation*, as shown in Figure 4.7.

A horizontal computation can be potentially faster than an equivalent number of vertical computations. For example, a horizontal computation of sample x from M in E avoids the overhead of evaluating the same term M more than once; thus the advantage of a horizontal computation becomes pronounced if M takes a long time to evaluate. The cost associated with each language construct also remains constant in a horizontal computation. For example, a horizontal computation of sample x from M in E performs a substitution for x only once, but vertical computations perform as many substitutions for x.

To examine the potential benefit of horizontal computations, we implement a translator of PTP for horizontal computations. Conceptually an expression now computes to an ordered set of samples in such a way that each sample corresponds to the result of an independent vertical computation of the same expression. We may think of the translator as implementing an operational semantics based upon the judgment

$$E \mathbin{@} [\omega_1, \cdots, \omega_n] \rightharpoonup\!\!\!\rightharpoonup \{V_1, \cdots, V_n\} \mathbin{@} [\omega_1', \cdots, \omega_n']$$

which means $E \mathbin{@} \omega_i \rightharpoonup V_i \mathbin{@} \omega_i'$ for $1 \leq i \leq n$.

The translator is implemented in a similar way to the operational semantics for vertical computations: the syntax of Objective CAML is extended using CAMLP4, and terms and expressions of the extended syntax are translated back in Objective CAML. The definition of the type constructor `prob`, however, is more complex because of conditional constructs (if $\cdot$ then $\cdot$ else $\cdot$ and eif $\cdot$ then $\cdot$ else $\cdot$). To motivate our definition of `prob`, consider the following expression:

sample x from prob $\mathcal{S}$ in
sample y from prob E in
eif $x \leq 0.5$ then E_1 else E_2

A vertical computation reduces the whole expression to either E_1 or E_2 and needs to keep only one reduced expression. A horizontal computation, however, may have to keep both E_1 and E_2 because multiple samples are generated from $U(0.0, 1.0]$ for variable x. For example, if an ordered set $\{0.1, 0.6, 0.3, 0.9\}$ is generated for variable x, the horizontal computation reduces to two smaller horizontal computations: one of E_1 with x bound to $\{0.1, -, 0.3, -\}$ and another of E_2 with x bound to $\{-, 0.6, -, 0.9\}$. Note that we may not compress $\{0.1, -, 0.3, -\}$ to $\{0.1, 0.3\}$ and $\{-, 0.6, -, 0.9\}$ to $\{0.6, 0.9\}$ because the ordered set to which variable y is bound may be correlated to variable x.

Thus we are led to define the type constructor `prob` using bit vectors and ordered sets:

```
type bflag
type 'a oset
type 'a prob = bflag -> 'a oset
```

- `bflag` is the type of bit vectors of fixed size.
- `'a oset` is a datatype for ordered sets of element type `'a`. An ordered set of element type `'a` may contain not only ordinary values of type `'a` but also *null values* ('$-$' in the above example). Ordinary values correspond to values of 1 and null values to values of 0 in bit vectors.
- `'a prob` is a datatype for both probability distributions over type `'a` and expressions of type `'a`. It is defined as the type of a function that takes a bit vector, performs a horizontal computation for values of 1 in the given bit vector, and returns the resultant ordered set.

Since variables from bind expressions are always bound to ordered sets, we distinguish between terms manipulating ordinary values and terms manipulating ordered sets. The new syntax, further augmenting the extended syntax in Section 4.3, introduces a non-terminal $\langle pterm \rangle$ for those terms manipulating ordered sets; the definition of the non-terminal $\langle expr \rangle$ uses $\langle pterm \rangle$ in place of $\langle term \rangle$:

$\langle term \rangle$	::=	$\cdots \mid \langle pterm \rangle$	
$\langle pterm \rangle$	::=	`lam` $\langle patt \rangle$ `->` $\langle pterm \rangle \mid$	lambda abstraction
		`app` $\langle pterm \rangle$ `to` $\langle pterm \rangle \mid$	application term
		`pif` $\langle pterm \rangle$ `then` $\langle pterm \rangle$ `else` $\langle pterm \rangle \mid$	cond. term construct
		`@`$\langle id \rangle \mid$	variable
		`const` $\langle term \rangle \mid$	constants
		`ptrue` $\mid$ `pfalse` $\mid$ `@+` $\mid$ `CMP <=.` $\mid \cdots$	built-in constants

In the new syntax, a Bernoulli distribution and a geometric distribution are encoded as follows:

```
let bernoulli = fun p ->
  PROB { sample x from PROB { UNIFORM } in
         [pif @x CMP <=. const p then ptrue else pfalse] }

let geometric = fun p ->
  let bernoulli_p = bernoulli_prob p in
  PROB {
    efix geo ->
      sample b from bernoulli_p in
      eif @b then [const 0]
      else
        sample x from PROB { #geo } in
        [const 1 @+ @x]
  }
```

Compared with the examples in Section 4.3, the code is the same except that all terms within expressions manipulate ordered sets rather than ordinary values.

test case	vertical	horizontal	overhead (%)
$bernoulli$ 0.25	0.922	1.188	28.85
$uniform$ 0.0 1.0	0.906	1.078	18.98
$binomial$ 0.25 16	16.563	23.187	39.99
$geometric\_efix$ 0.25	3.937	7.157	81.78
$gaussian\_rejection$ 2.5 5.0	4.688	7.593	61.96
$exponential\_von\_Neumann_{1.0}$	4.031	6.922	71.71
$gaussian\_Box\_Muller$ 2.0 4.0	4.796	5.031	4.89
$gaussian\_central$ 0.0 1.0	10.594	12.157	14.75
$Q_{Mary\_calls \mid John\_calls}$	90.063	138.922	54.24

Figure 4.8: Execution times (in seconds) for generating a total of 3,100,000 samples.

Experimental results

We compare execution times for generating the same number of samples in vertical and horizontal computations. The type `bflag` uses 31-bit integers (of type `int` in Objective CAML), which means that a single horizontal computation performs up to 31 independent vertical computations; the datatype `'a oset` uses arrays of 31 elements of type `'a`. We use an AMD Athlon XP 1.67GHz with 512MB memory for all experiments.

Figure 4.8 shows execution times for various test cases from Chapter 3. In all test cases, horizontal computations are slower than vertical computations, as indicated by their overhead relative to vertical computations. The overhead of horizontal computations is especially high in those test cases involving conditional constructs (namely, $binomial$, $geometric\_efix$, $gaussian\_rejection$, $exponential\_von\_Neumann_{1.0}$, and $Q_{Mary\_calls \mid John\_calls}$). The high overhead can be attributed to the fact that a horizontal computation allocates an array of size 31 for every expression, regardless of the number of ordinary values from it. For example, even when a horizontal computation is simulating just a single vertical computation (after encountering several conditional constructs), the computation of an expression still requires an array of size 31.

The experimental results show that the overhead for maintaining ordered sets and handling conditional constructs exceeds the gain from simulating multiple vertical computations with a single horizontal computation. Our implementation is just a translator which does not rely on support from the compiler. In order to fully realize the potential of horizontal computations, it seems necessary to integrate the implementation within the compiler and the run-time system. As a speculation, horizontal computations can be up to twice faster than vertical computations: random number generation, which costs the same in both vertical and horizontal computations, accounts for about half the total computation time; hence, with no overhead other than random number generation, horizontal computations would be about twice faster than vertical computations.

4.6 Summary

Although PTP is implemented indirectly via a translation in Objective CAML, both its type system and its operational semantics are faithfully mirrored through the use of an abstract datatype. Besides all existing features of Objective CAML are available when programming in PTP, and we may think of the implementation of PTP as a conservative extension of Objective CAML. The translation is easily generalized to any monadic language, thus complementing the well-established result that a call-by-value language is translated in a monadic language (*e.g.*, see [68]).

The translator of PTP does not protect terms from computational effects already available in Objective CAML such as input/output, mutable references, and even direct uses of `Random.float`. Thus, for example, term M in a bind expression $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$ is supposed to produce no world effect, but the translator has no way to verify that the evaluation of M is effect-free. Therefore the translator of PTP relies on programmers to ensure that every term denotes a regular value.

Since the linguistic framework for PTP is a reformulation of Moggi's monadic metalanguage λ_{ml} (see Chapter 2), Haskell is also a good choice as a host language for embedding PTP. To embed PTP in Haskell, one would define a Haskell monad, say `Prob`, for probabilistic choices and translate an expression of type A into a program fragment of type `Prob` A, while ignoring the keyword prob in probability terms. Alternatively one could exploit the global random number generator maintained by the `IO` monad and translate $\bigcirc A$ of PTP into `IO` A of Haskell. (Our choice of Objective CAML is due to personal preference.)

We could directly implement PTP by extending the compiler and the run-time system of Objective CAML. An immediate benefit is that type error messages are more informative because type errors are detected at the level of PTP. (Our implementation detects type errors in the translated code rather than in the source code; hence programmers should analyze type error messages to locate type errors in the source code.) As for execution speed, we conjecture that the gain is negligible, since the only overhead incurred by the abstract datatype `prob` is to invoke two tiny functions when its member functions are invoked: an identity function (for `prb`) and a function applying its argument to a unit value (for `app`).

Chapter 5

Applications

This chapter presents three applications of PTP in robotics: robot localization, people tracking, and robotic mapping, all of which are popular topics in robotics. Although different in goal, all these applications share a common characteristic: the state of a robot is estimated from sensor readings, where the definition of state differs in each case. A key element of these applications is uncertainty in sensor readings, due to limitations of sensors and noise from the environment. It makes the problem of estimating the state of a robot both interesting and challenging: if all sensor readings were accurate, the state of a robot could be accurately traced by a simple (non-probabilistic) analysis of sensor readings. In order to cope with uncertainty in sensor readings, we estimate the state of a robot with probability distributions.

As a computational framework, we use Bayes filters. In each case, we formulate the update equations at the level of probability distributions and translate them in PTP. All implementations are tested using data collected with real robots.

5.1 Sensor readings: action and measurement

To update the state of a robot, we use two kinds of sensor readings: *action* and *measurement*. As in a Bayes filter, an action induces a state change whereas a measurement gives information on the state:

- An action a is represented as an odometer reading which returns the pose (*i.e.*, position (x, y) and orientation θ) of the robot relative to its initial pose. It is given as a tuple $(\Delta x, \Delta y, \Delta\theta)$.
- A measurement m consists of range readings which return distances to objects visible at certain angles. It is given as an array $[d_1; \cdots ; d_n]$ where each d_i, $1 \leq i \leq n$, denotes the distance between the robot and the closest object visible at a certain angle.

Figure 5.1 shows a typical example of measurement. It displays range readings produced by a laser range finder covering 180 degrees. The robot is shown in the center; occluded regions are colored in grey.

Odometers and range finders are prone to errors because of their mechanical nature. An odometer usually tends to drift in one direction over time. Its accumulated error becomes manifest especially when the robot closes a loop after taking a circular route. Range finders occasionally fail to recognize obstacles and report the maximum distance measurable. In order to correct these errors, we use a probabilistic approach by representing the state of the robot with a probability distribution.

In the probabilistic approach, an action increases the set of possible states of the robot because it induces a state change probabilistically. In contrast, a measurement decreases the set of possible states of the robot because it gives negative information on unlikely states (and positive information on likely states). We now demonstrate how to probabilistically update the state of the robot in three different applications.

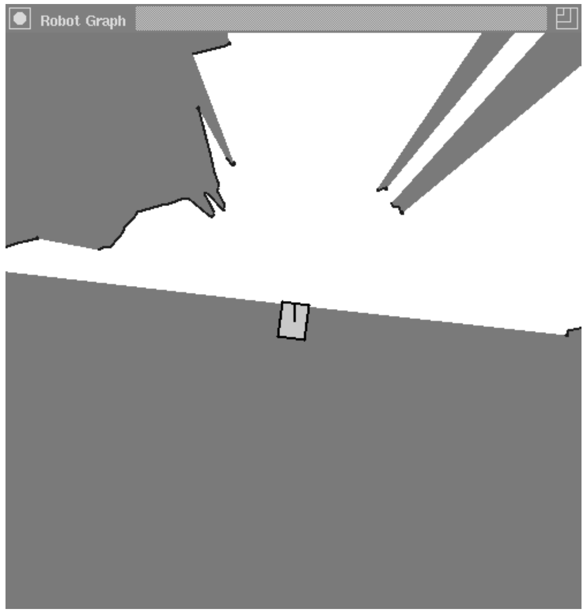


Figure 5.1: Range readings produced by a laser range finder. The robot faces a person on its right, visible as the shadows of two legs.

5.2 Robot localization

Robot localization [72] is the problem of estimating the pose of a robot when a map of the environment is available. If the initial pose is given, the problem becomes *pose tracking* which keeps track of the robot pose by compensating errors in sensor readings. If the initial pose is not given, the problem becomes *global localization* which begins with multiple hypotheses on the robot pose (and is therefore more involved than pose tracking).

We consider robot localization under the assumption (called the *Markov assumption*) that the past and the future are independent if the current pose is known, or equivalently that the environment is static. This assumption allows us to use a Bayes filter in estimating the robot pose. Specifically the state in the Bayes filter is the robot pose $s = (x, y, \theta)$, and we estimate s with a probability distribution $Bel(s)$ over three-dimensional real space. We compute $Bel(s)$ according to the following update equations (which are the same as shown in Section 1.1):

$$Bel(s) \leftarrow \int \mathcal{A}(s|a, s')Bel(s')ds' \tag{5.1}$$

$$Bel(s) \leftarrow \eta \mathcal{P}(m|s)Bel(s) \tag{5.2}$$

η a normalizing constant ensuring $\int Bel(s)ds = 1.0$. We use the following interpretation of $\mathcal{A}(s|a, s')$ and $\mathcal{P}(m|s)$:

- $\mathcal{A}(s|a, s')$ is the probability that the robot moves to pose s after taking action a at another pose s'. $\mathcal{A}$ is called an *action model*.
- $\mathcal{P}(m|s)$ is the probability that measurement m is taken at pose s. $\mathcal{P}$ is called a *perception model*.

Given an action a and a pose s', a new pose s can be generated from the action model $\mathcal{A}(\cdot|a, s')$ by adding a noise to a and applying it to s'. In our implementation, $\mathcal{A}(\cdot|a, s')$ assumes constant translational and rotational velocities while action a is taken from pose s'. It also assumes that errors in translational and

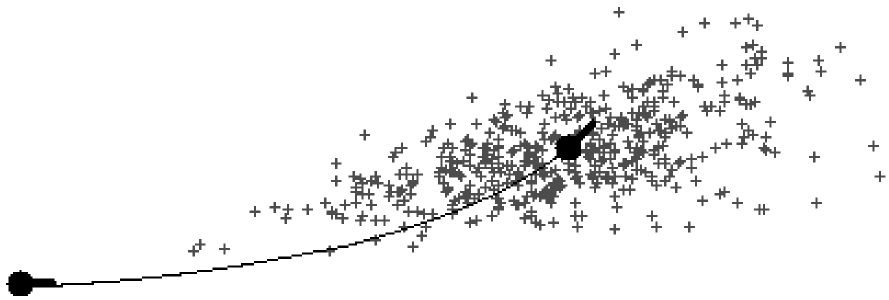


Figure 5.2: Samples from the action model.

rotational velocities obey Gaussian distributions. Figure 5.2 shows samples of the new pose after taking a curved trajectory.

Given a measurement m and a pose s, we can also compute $\kappa\mathcal{P}(m|s)$ where κ is an unknown constant: the map determines a unique (accurate) measurement m_s for pose s, and the squared Euclidean distance between m and m_s is assumed to be proportional to $\mathcal{P}(m|s)$. Figures 5.3 and 5.4 illustrate how to compute $\kappa\mathcal{P}(m|s)$. Figure 5.3 shows points in the map that correspond to measurement m when s is set to the true pose of the robot, in which case the unique measurement m_s for pose s coincides with m (recall that a measurement consists of not points in the map but range readings). Hence each point is projected on the contour of the map and is assigned a high likelihood as indicated by the dark color. Figure 5.4 shows points in the map that correspond to the same measurement m, but when s is set to a hypothetical pose of the robot; the unique measurement m_s for pose s is represented by points with crosses. Since the measurement is not taken at the hypothetical pose, no point is correctly aligned along the contour of the map. Thus each point is assigned a relatively low likelihood as indicated by the grey color (the degree of darkness indicates its likelihood). We compute $\kappa\mathcal{P}(m|s)$ as the product of all individual likelihoods.[1]

Our implementation simplifies the computation of $\kappa\mathcal{P}(m|s)$ by approximating m_s with those points on the contour of the map that are closest to the points corresponding to measurement m; Figure 5.5 shows how to approximate m_s with those points with crosses. This simplification allows us to precompute the likelihood of every point in the map, since its closest point on the contour of the map is fixed. Our implementation uses a grid map at 10 centimeter resolution and generates a *likelihood map* which stores the likelihood of each cell in the map; see Figures 5.6 for a grid map and its likelihood map.

Now, if $M_{\mathcal{A}}$ denotes conditional probability $\mathcal{A}$ and $M_{\mathcal{P}}\ m$ returns a function $f(s) = \kappa\mathcal{P}(m|s)$, we implement update equations (5.1) and (5.2) as follows:

$$\begin{array}{lll} \mathsf{let}\ Bel_{new} = \mathsf{prob} & \mathsf{sample}\ s'\ \mathsf{from}\ Bel\ \mathsf{in} & \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} (5.1) \\ & \mathsf{sample}\ s\ \mathsf{from}\ M_{\mathcal{A}}\ (a, s')\ \mathsf{in} & \\ & s & \\ \mathsf{let}\ Bel_{new} = \mathsf{bayes}\ (M_{\mathcal{P}}\ m)\ Bel & & \}\ (5.2) \end{array}$$

Both pose tracking and global localization are achieved by specifying an appropriate initial probability distribution of robot pose. For pose tracking, we use a point-mass distribution or a Gaussian distribution;

[1]Our implementation filters out outlier range readings in m before computing $\kappa\mathcal{P}(m|s)$.

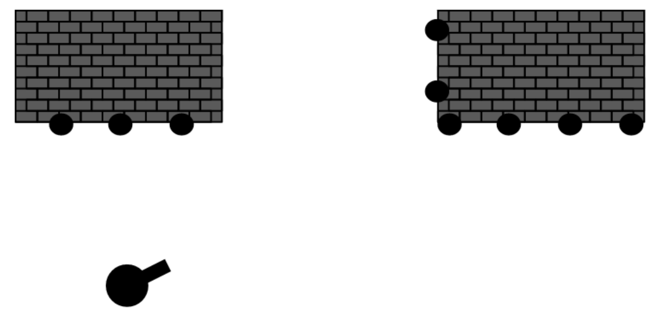


Figure 5.3: Points in the map that correspond to measurements when s is set to the true pose of the robot.

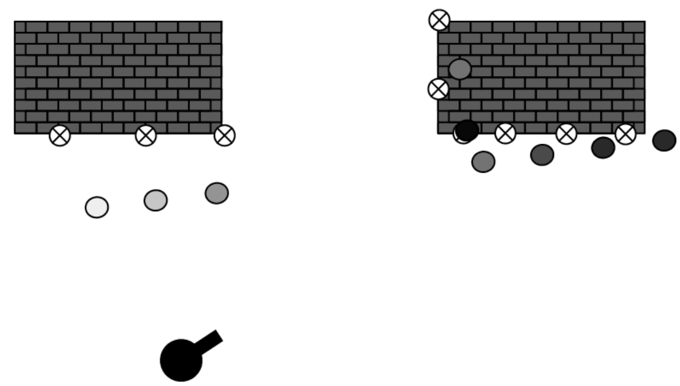


Figure 5.4: Points in the map that correspond to measurements when s is set to a hypothetical pose of the robot.

for global localization, we use a uniform distribution over the open space in the map.

Experimental results

To test the robot localizer, we use a Nomad XR4000 mobile robot in Wean Hall at Carnegie Mellon University. The robot is equipped with 180 laser range finders (one for each degree so as to cover 180 degrees). The robot localizer uses every fifth range reading, and thus a measurement consists of a batch of $\frac{180}{5} = 36$ range readings. We use CARMEN [49] for controlling the robot and collecting sensor readings. The robot localizer runs on a Pentium III 500Mhz with 384 MBytes memory.

We test the robot localizer for global localization. The initial probability distribution of robot pose is a uniform distribution over the open space in the map, which is approximated with 100,000 samples. The first batch of range readings is processed according to update equation (5.2). The resultant probability distribution, which is still approximated with 100,000 samples, is then replaced by its support approximated with 500 samples. The number of samples, 100,000 or 500, is chosen empirically — both too many and too few samples prevent the probability distribution from converging to a correct pose.

Figure 5.7 shows a probability distribution of robot pose after processing the first batch of range readings in Figure 5.1; pluses represent samples generated from the probability distribution. The robot starts right below character A, but there are relatively few samples around the true position of the robot. Figure 5.8 shows the progress of a real-time robot localization run that continues with the probability distribution in Figure 5.7. The first two pictures show that the robot localizer is still performing global localization. The last picture shows that the robot localizer has started pose tracking as the probability distribution of robot

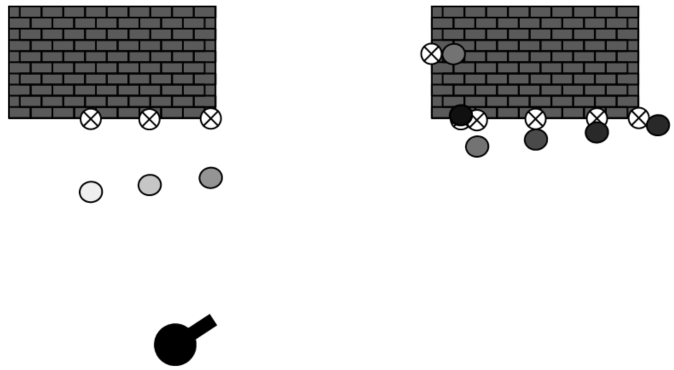


Figure 5.5: Approximating m_s from measurement m and pose s.

pose has converged to a single hypothesis.

We test the robot localizer with 8 runs, each of which takes a different path. In a test experiment, it succeeds to localize the robot on 5 runs and fails on 3 runs. (The result should not be considered statistically significant.) As a comparison, the CARMEN robot localizer, which uses particle filters and is written in C, succeeds on 3 runs and fails on 5 runs under the same condition (100,000 samples during initialization, 500 samples during localization, and 36 range readings in each measurement). Note that the same sequence of sensor readings does not guarantee the same result because of the probabilistic nature of the robot localizer. In the worst scenario, for example, the initial probability distribution of robot pose may have no samples around the true pose, in which case the robot localizer is unlikely to recover from errors. Hence it is difficult to precisely quantify the performance of the robot localizer; the goal is to convince that our implementation in PTP is reasonably acceptable, not totally fake.

5.3 People tracking

People tracking [50] is an extension of robot localization in that it estimates not only the robot pose but also positions of people (or unmapped objects). As in robot localization, the robot takes an action to change its pose. Unlike in robot localization, however, the robot categorizes sensor readings in a measurement by deciding whether they correspond with objects in the map or with people. Those sensor readings that correspond with objects in the map are used to update the robot pose; the rest of sensor readings are used to update positions of people.

A simple approach is to maintain a probability distribution $Bel(s, \vec{u})$ of robot pose s and positions $\vec{u}$ of people. Although it works well for pose tracking, this approach is not a general solution for global localization. The reason is that sensor readings from people are correctly interpreted only with a correct hypothesis on the robot pose, but during global localization, there may be incorrect hypotheses that lead to incorrect interpretation of sensor readings. For example, the two objects in the upper right region in Figure 5.1 are interpreted as a person only with a correct hypothesis on the robot pose. This means that during global localization, there exists a dependence between the robot pose and positions of people, which is not captured by $Bel(s, \vec{u})$.

Hence we maintain a probability distribution $Bel(s, P_s(\vec{u}))$ of robot pose s and *probability distribution $P_s(\vec{u})$ of positions $\vec{u}$ of people conditioned on robot pose s.*[2] $P_s(\vec{u})$ captures the dependence between the

[2]Our implementation assumes that people move independently of each other, and represents $P_s(\vec{u})$ as a set of independent probability distributions each of which keeps track of the position of an individual person.

Figure 5.6: A grid map and its likelihood map.

robot pose and positions of people. $Bel(s, P_s(\vec{u}))$ can be thought of as a probability distribution over probability distributions.

As in robot localization, we update $Bel(s, P_s(\vec{u}))$ with a Bayes filter. The difference from robot localization is that the state is a pair of s and $P_s(\vec{u})$ and that the action model takes as input both an action a and a measurement m. We use update equations (5.3) and (5.4) in Figure 5.9 (which are obtained by replacing s by $s, P_s(\vec{u})$ and a by a, m in update equations (1.1) and (1.2)).

The action model $\mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}))$ generates $s, P_s(\vec{u})$ from $s', P_{s'}(\vec{u'})$ utilizing action a and measurement m. We first generate s and then $P_s(\vec{u})$ according to equation (5.5) in Figure 5.9. We write the first $Prob$ in equation (5.5) as $\mathcal{A}_{\textsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$. The second $Prob$ in equation (5.5) indicates that we generate $P_s(\vec{u})$ from $P_{s'}(\vec{u'})$ utilizing action a and measurement m, which is exactly a situation where we can use another Bayes filter. For this inner Bayes filter, we use update equations (5.6) and (5.7) in Figure 5.9. We write $Prob$ in equation (5.6) as $\mathcal{A}_{\textsf{people}}(\vec{u}|a, \vec{u'}, s, s')$; we simplify $Prob$ in equation (5.7) into $Prob(m|\vec{u}, s)$ because m does not depend on s' if s is given, and write it as $\mathcal{P}_{\textsf{people}}(m|\vec{u}, s)$.

Figure 5.10 shows the implementation of people tracking in PTP. $M_{\mathcal{A}_{\textsf{robot}}}$ and $M_{\mathcal{A}_{\textsf{people}}}$ denote conditional probabilities $\mathcal{A}_{\textsf{robot}}$ and $\mathcal{A}_{\textsf{people}}$, respectively. $M_{\mathcal{P}_{\textsf{people}}}\ m\ s$ returns a function $f(\vec{u}) = \kappa \mathcal{P}_{\textsf{people}}(m|\vec{u}, s)$ for a constant κ. Since both m and s are fixed when computing $f(\vec{u})$, we consider only those range readings in m that correspond with people. In implementing update equation (5.4), we use the fact that $\mathcal{P}(m|s, P_s(\vec{u}))$ is the expectation of a function $g(\vec{u}) = \mathcal{P}_{\textsf{people}}(m|\vec{u}, s)$ with respect to $P_s(\vec{u})$:

$$\mathcal{P}(m|s, P_s(\vec{u})) = \int \mathcal{P}_{\textsf{people}}(m|\vec{u}, s) P_s(\vec{u}) d\vec{u} \tag{5.8}$$

Our implementation further simplifies the models used in the update equations. We use $\mathcal{A}_{\textsf{robot}}(s|a, s')$ instead of $\mathcal{A}_{\textsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$ as in robot localization. That is, we ignore the interaction between the robot and people when generating new poses of the robot. Similarly we use $\mathcal{A}_{\textsf{people}}(\vec{u}|\vec{u'})$ instead of

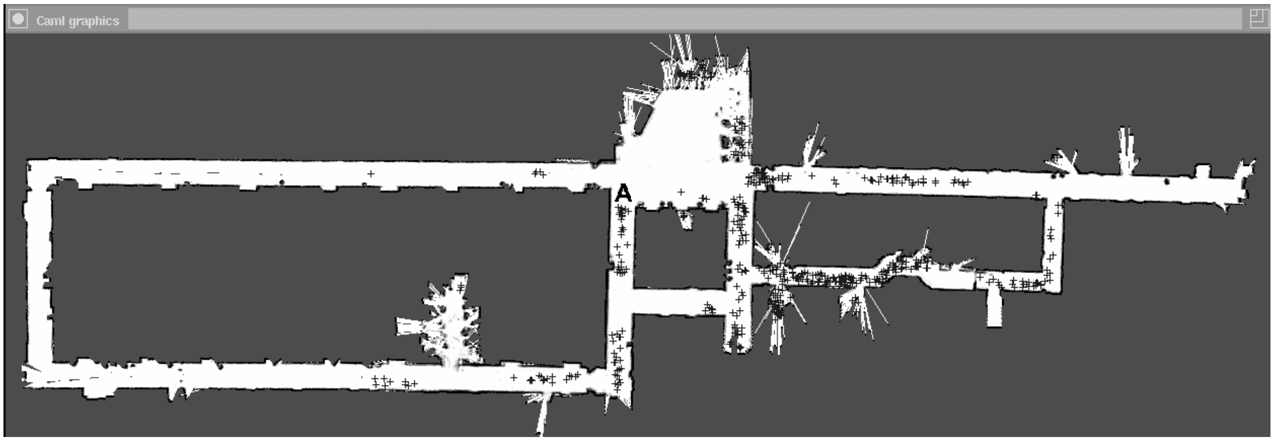


Figure 5.7: Probability distribution of robot pose after processing the first batch of range readings in Figure 5.1.

$\mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')$ on the assumption that positions of people are not affected by the robot pose; $\vec{u}$ is obtained by adding a random noise to $\vec{u'}$. We also simplify $\mathcal{P}(m|s, P_s(\vec{u}))$ in update equation (5.4) into $\mathcal{P}(m|s)$, which is computed in the same way as in robot localization; hence equation (5.8) is not actually exploited in our implementation.

Experimental results

We test the people tracker on the same robot and machine that are used in robot localization. The people tracker uses the implementation in Figure 5.10 during global localization, but once it succeeds to localize the robot and starts pose tracking, it maintains a probability distribution $Bel(s, \vec{u})$ as there is no longer a dependence between the robot pose and positions of people. Like the robot localizer, we do not intend to quantitatively measure the success rate of people tracking; rather the focus is on ensuring that our implementation in PTP is not completely useless.

Figure 5.11 shows the progress of a real-time people tracking run which uses the same sequence of sensor readings as Figure 5.8. The first picture is taken after processing the first batch of range readings in Figure 5.1; pluses ($+$) represent robot poses and crosses ($\times$) represent positions of people. The second picture shows that the people tracker is still performing global localization. The last picture shows that the people tracker has started pose tracking; the position of each person in sight is indicated by a grey dot. Figure 5.12 shows range readings when the third picture in Figure 5.11 is taken; the right picture shows a magnified view of the area around the robot. Note that a person may be occluded by another person or objects in the map, so grey dots do not always reflect the movement of people instantly. A refined action model for people (*e.g.*, $\mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')$ or one estimating not only the position but also the velocity of each person) would alleviate the problem.

5.4 Robotic mapping

Robotic mapping [75] is the problem of building a map (or a spatial model) of the environment from sensor readings. Since measurements are a sequence of inaccurate local snapshots of the environment, a robot simultaneously localizes itself as it explores the environment so that it corrects and aligns local snapshots to construct a global map. For this reason, robotic mapping is also referred to as *simultaneous localization and mapping* (or SLAM).

Taking a probabilistic approach, we formulate the robotic mapping problem with a Bayes filter which maintains a probability distribution $Bel(s,g)$ of robot pose s and map g. Given an action a and a measurement m, we update $Bel(s,g)$ as follows:

$$Bel(s,g) \;\leftarrow\; \int_{s',g'} \mathcal{A}(s,g|a,s',g')Bel(s',g')d(s',g') \tag{5.9}$$

$$Bel(s,g) \;\leftarrow\; \eta\mathcal{P}(m|s,g)Bel(s,g) \tag{5.10}$$

We assume that an action is independent of the map and does not change the environment; that is, $\mathcal{A}(s,g|a,s',g')$ $= \mathcal{A}(s|a,s')$ if $g = g'$, and $\mathcal{A}(s,g|a,s',g') = 0$ if $g \neq g'$. Then we can simplify update equation (5.9) as follows:

$$Bel(s,g) \;\leftarrow\; \int_{s'} \mathcal{A}(s|a,s')Bel(s',g)ds' \tag{5.11}$$

Therefore the action model becomes the same as in robot localization. We implement the new update equation (5.11) as follows:

let $Bel_{new} =$ prob sample (s',g) from Bel_{old} in sample s from $M_{\mathcal{A}}\ (a,s')$ in (s,g)

The update equation (5.10) is implemented with a Bayes operation as before.

Unfortunately the space of maps has a huge dimension, which makes it impossible to maintain $Bel(s,g)$ without simplifying their representation. Therefore we usually make additional assumptions on maps to derive a specific representation. For example, assuming that a map consists of a set of landmarks whose locations are estimated with Gaussian distributions, we can use a Kalman filter instead of a general Bayes filter. If measurements, or local snapshots of the environment, are assumed to be accurate relative to robot poses, we can represent a map by the sequence of robot poses when the measurements are taken, as in [38]. We can also exploit expectation maximization [14], in which we perform hill climbing in the space of maps to find the most likely map. This approach does not maintain a probability distribution over maps because it keeps only one (most likely) map at each iteration.

Here we assume that the environment consists of an unknown number of stationary landmarks. Then the goal is to estimate positions of landmarks as well as the robot pose. The key observation is that we may think of landmarks as people who never move in an empty environment. It means that the problem is a special case of people tracking and we can use all the equations in Figure 5.9. Below we use subscript $_{\mathsf{landmark}}$ instead of $_{\mathsf{people}}$ for the sake of clarity.

As in people tracking, we maintain a probability distribution $Bel(s, P_s(\vec{u}))$ of robot pose s and probability distribution $P_s(\vec{u})$ of positions $\vec{u}$ of landmarks conditioned on robot pose s. Since landmarks are stationary and $\mathcal{A}_{\mathsf{landmark}}(\vec{u}|a,\vec{u'},s,s')$ is non-zero if and only if $\vec{u} = \vec{u'}$, we skip update equation (5.6) in implementing update equation (5.3). $\mathcal{A}_{\mathsf{robot}}$ in equation (5.5) uses $\mathcal{P}_{\mathsf{landmark}}(m|\vec{u'},s)$ to test the likelihood of each new robot pose s with respect to old positions $\vec{u'}$ of landmarks, as in FastSLAM 2.0 [48]:

$$\begin{array}{rl} & \mathcal{A}_{\mathsf{robot}}(s|a,m,s',P_{s'}(\vec{u'})) \\ = & \int Prob(s|a,m,s',u')P_{s'}(\vec{u'})d\vec{u'} \\ = & \displaystyle\int \frac{Prob(s|a,\vec{u'})Prob(m,s'|s,a,\vec{u'})}{Prob(m,s'|a,\vec{u'})} P_{s'}(\vec{u'})d\vec{u'} \\ = & \int \eta'' Prob(m,s'|s,a,\vec{u'})P_{s'}(\vec{u'})d\vec{u'} \quad where \quad \eta'' = \dfrac{Prob(s|a,\vec{u'})}{Prob(m,s'|a,\vec{u'})} \\ = & \int \eta'' Prob(s'|s,a,\vec{u'},m)Prob(m|s,a,\vec{u'})P_{s'}(\vec{u'})d\vec{u'} \\ = & \int \eta'' Prob(s'|s,a)Prob(m|s,\vec{u'})P_{s'}(\vec{u'})d\vec{u'} \\ = & \eta''\mathcal{A}_{\mathsf{robot}}(s|a,s')\int \mathcal{P}_{\mathsf{landmark}}(m|\vec{u'},s)P_{s'}(\vec{u'})d\vec{u'} \end{array} \tag{5.12}$$

Given a and s', we implement equation (5.12) with a Bayes operation on $\mathcal{A}_{\mathsf{robot}}(\cdot|a, s')$.

Figure 5.13 shows the implementation of robotic mapping in PTP. Compared with the implementation of people tracking in Figure 5.10, it omits update equation (5.6) and incorporates equation (5.12). $M_{\mathcal{A}_{\mathsf{robot}}}$ and $M_{\mathcal{P}_{\mathsf{landmark}}}$ denote conditional probabilities $\mathcal{A}_{\mathsf{robot}}$ and $\mathcal{P}_{\mathsf{landmark}}$, respectively, as in people tracking. Since landmarks are stationary, we no longer need $M_{\mathcal{A}_{\mathsf{landmark}}}$. If we approximate $Bel(s, P_s(\vec{u}))$ with a single sample (*i.e.*, with one most likely robot pose and an associated map), update equation (5.4) becomes unnecessary.

Experimental results

To test the mapper, we use a data set collected with an outdoor vehicle in Victoria Park, Sydney [1]. The mapper runs on the same machine that is used in robot localization and people tracking (Pentium III 500Mhz with 384 MBytes memory). The data set is collected while the vehicle moves approximately 323.42 meters (according to the odometry readings) in 128.8 seconds. Since the vehicle is driving over uneven terrain, raw odometry readings are noisy and do not reflect the true path of the vehicle, in particular when the vehicle follows a loop.

Figure 5.14 shows raw odometry readings in the data set. The true positions of the vehicle measured by a GPS sensor are represented by crosses, which are available only for part of the entire traverse and are not exploited by the mapper. Note that the odometry readings eventually diverge from the true path of the vehicle. Figure 5.15 shows the result of the robotic mapping experiment in which we approximate $Bel(s, P_s(\vec{u}))$ with a single sample and use 1,000 samples for the expectation query and the Bayes operation. The circles represent landmark positions (mean of their probability distributions). The mapper successfully closes the loop, building a map of the landmarks around the path. The experiment, however, takes 145.89 seconds, which is 13.26% longer than it takes to collect the data set (128.8 seconds).

5.5 Summary

PTP is a probabilistic language which allows programmers to concentrate on how to formulate probabilistic computations at the level of probability distributions, regardless of the kind of probability distributions involved. The three applications in robotics substantiate the practicality of PTP by illustrating how to directly translate a probabilistic computation into code and providing experimental results on real robots.

Our finding is that the benefit of implementing probabilistic computations in PTP, such as improved readability and conciseness of code, can outweigh its disadvantage in speed. For example, our robot localizer is 1307 lines long (826 lines of Objective CAML/PTP code for probabilistic computations and 481 lines of C code for interfacing with CARMEN) whereas the CARMEN robot localizer, which uses particle filters and is written in C, is 3397 lines long. (Our robot localizer also uses the translator of PTP which is 306 lines long: 53 lines of CAMLP4 code and 253 lines of Objective CAML code.) The comparison is, however, not conclusive because not every piece of code in CARMEN contributes to robot localization. Moreover the reduction in code size is also attributed to the use of Objective CAML as the host language. Hence the comparison should not be taken as indicative of reduction in code size due to PTP alone. The speed loss is also not significant. For example, while the CARMEN robot localizer processes 100.0 sensor readings, our robot localizer processes on average 54.6 sensor readings (and nevertheless shows comparable accuracy).

On the other hand, PTP does not allow programmers to exploit a particular representation scheme for probability distributions, which is inevitable for achieving high scalability in some applications. In the robotic mapping problem, for example, one may choose to approximate the position of each landmark with a Gaussian distribution. As the cost of representing a Gaussian distribution is relatively low, the approximation makes it possible to build a highly scalable mapper. For example, Montemerlo [48] presents a FastSLAM

2.0 mapper which handles maps with over 1,000,000 landmarks. For such a problem, PTP would be useful for quickly building a prototype implementation to test the correctness of a probabilistic computation.

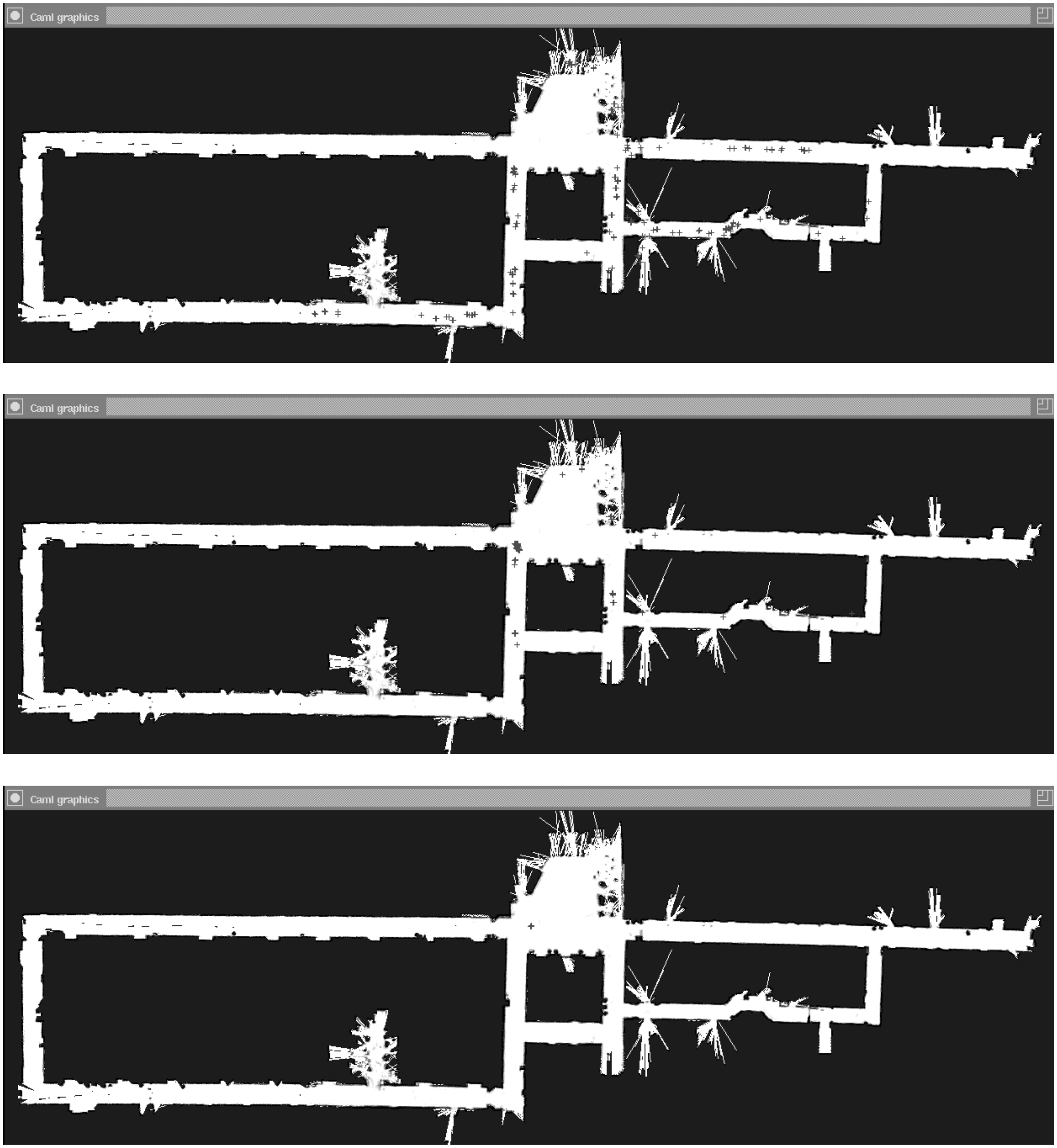


Figure 5.8: Progress of a real-time robot localization run. Taken at 20 seconds, 40 seconds, and 80 seconds after processing the first batch of sensor readings in Figure 5.1.

$$
\begin{array}{lrcl}
(5.3) & Bel(s, P_s(\vec{u})) & \leftarrow & \int \mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}))Bel(s', P_{s'}(\vec{u'}))d(s', P_{s'}(\vec{u'})) \\
(5.4) & Bel(s, P_s(\vec{u})) & \leftarrow & \eta\mathcal{P}(m|s, P_s(\vec{u}))Bel(s, P_s(\vec{u})) \\
 & & = & \eta Bel(s, P_s(\vec{u}))\int \mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)P_s(\vec{u})d\vec{u} \\
(5.5) & \mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'})) & = & Prob(s|a, m, s', P_{s'}(\vec{u'}))\ Prob(P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}), s) \\
 & & = & \mathcal{A}_{\mathsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))\ Prob(P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}), s) \\
(5.6) & P_s(\vec{u}) & \leftarrow & \int Prob(\vec{u}|a, \vec{u'}, s, s')P_{s'}(\vec{u'})d\vec{u'} \\
 & & = & \int \mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')P_{s'}(\vec{u'})d\vec{u'} \\
(5.7) & P_s(\vec{u}) & \leftarrow & \eta' Prob(m|\vec{u}, s, s')P_s(\vec{u}) \\
 & & = & \eta' \mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)P_s(\vec{u})
\end{array}
$$

Figure 5.9: Equations used in people tracking. (5.3) and (5.4) for the Bayes filter computing $Bel(s, P_s(\vec{u}))$. (5.5) for decomposing the action model. (5.6) and (5.7) for the inner Bayes filter computing $P_s(\vec{u})$.

$$
\begin{array}{l}
\mathsf{let}\ Bel_{new} = \\
\quad \mathsf{prob}\ \ \mathsf{sample}\ (s', P_{s'}(\vec{u'}))\ \mathsf{from}\ Bel\ \mathsf{in} \\
\qquad\quad \mathsf{sample}\ s\ \mathsf{from}\ M_{\mathcal{A}_{\mathsf{robot}}}\ (a, m, s', P_{s'}(\vec{u'}))\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ P_s(\vec{u}) = \mathsf{prob}\ \ \mathsf{sample}\ \vec{u'}\ \mathsf{from}\ P_{s'}(\vec{u'})\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\quad\ \ \mathsf{sample}\ \vec{u}\ \mathsf{from}\ M_{\mathcal{A}_{\mathsf{people}}}\ (a, \vec{u'}, s, s')\ \mathsf{in} \quad (5.6) \\
\qquad\qquad\qquad\qquad\quad\ \ \vec{u} \\
\qquad\qquad \mathsf{in} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (5.5) \quad (5.3) \\
\qquad\quad \mathsf{let}\ P_s(\vec{u}) = \mathsf{bayes}\ (M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u})\ \mathsf{in} \quad (5.7) \\
\qquad\quad (s, P_s(\vec{u})) \\
\mathsf{let}\ Bel_{new} = \\
\quad \mathsf{bayes}\ \lambda(s, P_s(\vec{u})) : \_.\ (\mathsf{expectation}\ (M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u}))\ Bel \quad (5.4)
\end{array}
$$

Figure 5.10: Implementation of people tracking in PTP. Numbers on the right-hand side show corresponding equations in Figure 5.9.

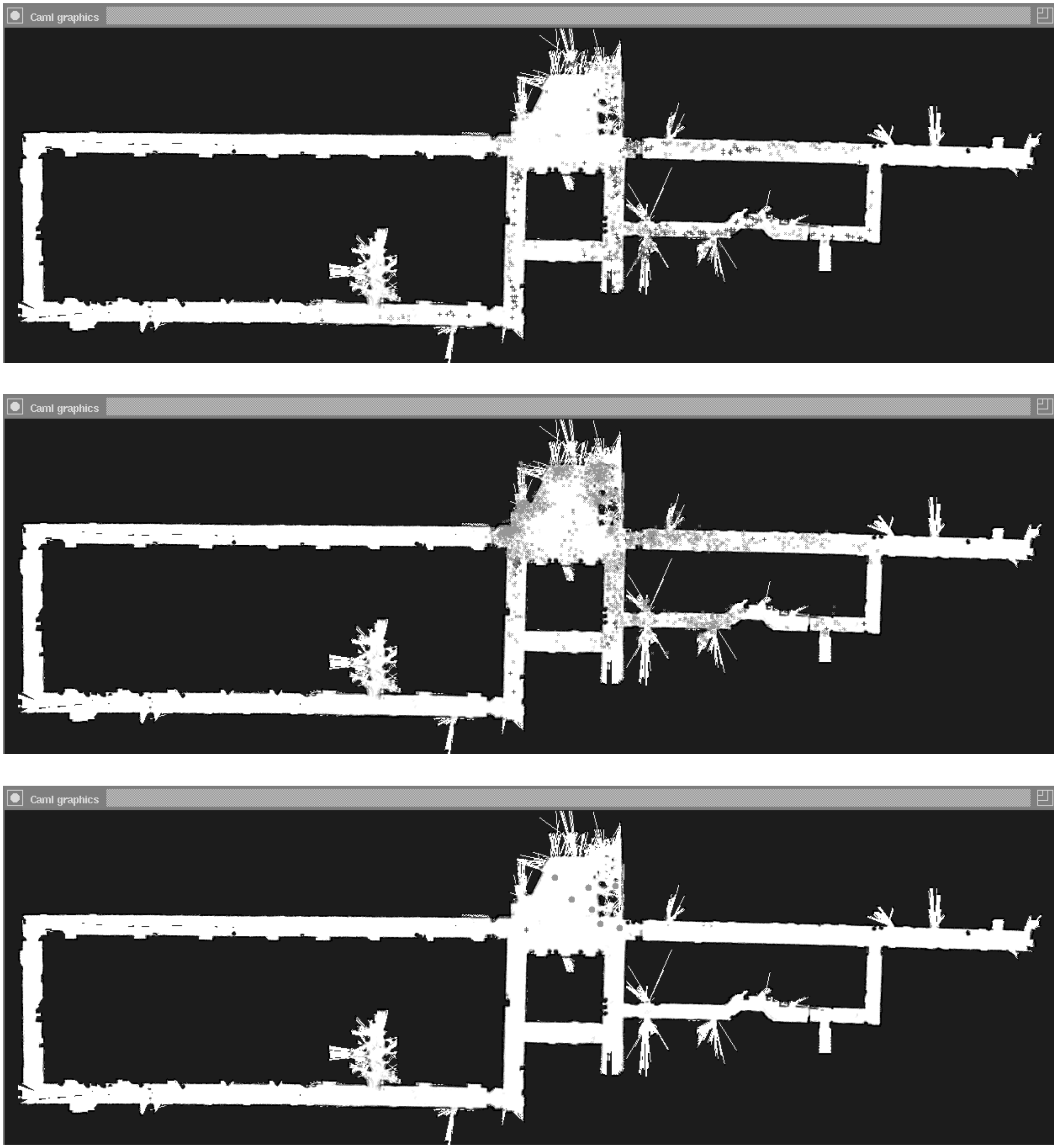


Figure 5.11: Progress of a real-time people tracking run. Taken at 0 seconds, 20 seconds, and 70 seconds after processing the first batch of sensor readings in Figure 5.1.

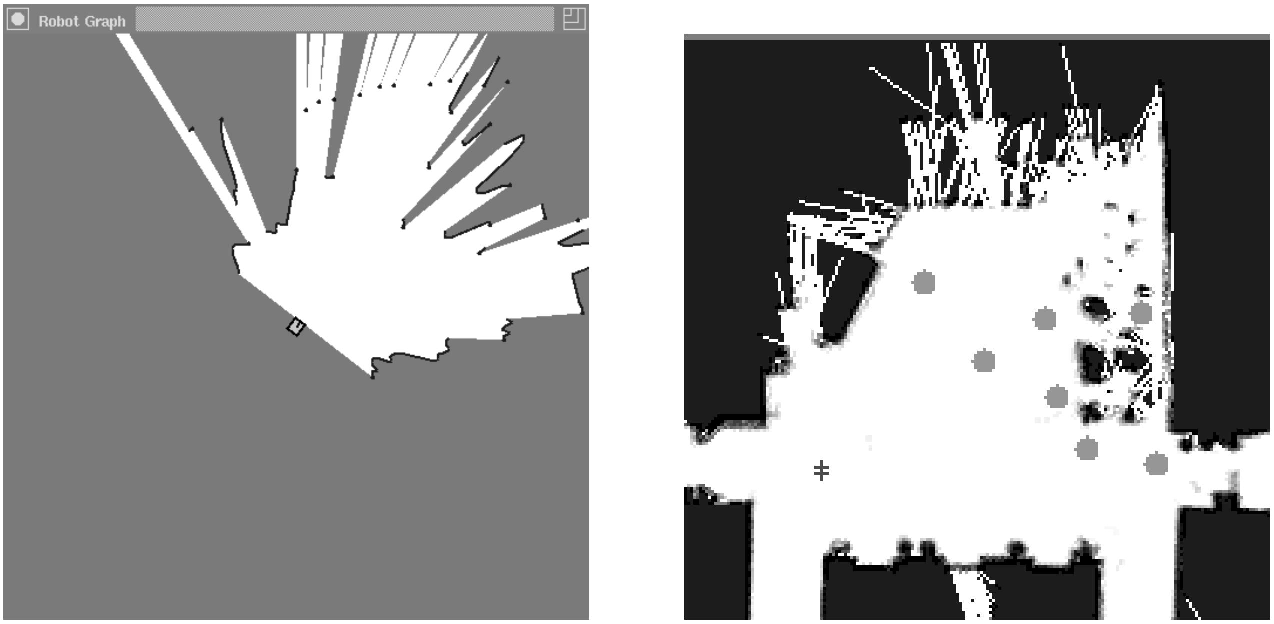


Figure 5.12: Range readings and the area around the robot during a people tracking run.

$$
\begin{array}{l}
\left.\begin{array}{l}
\text{let } Bel_{new} = \\
\quad \text{prob}\ \ \text{sample } (s', P_{s'}(\vec{u'})) \text{ from } Bel \text{ in} \\
\qquad \left.\begin{array}{l}
\left.\begin{array}{l}
\text{sample } s \text{ from} \\
\quad \text{bayes } \lambda s\!:\!\_.\ (\text{expectation } (M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_{s'}(\vec{u'})) \\
\qquad (M_{\mathcal{A}_{\text{robot}}}\ (a, s')) \text{ in}
\end{array}\right\}(5.12) \\
\left.\begin{array}{l}
\text{let } P_s(\vec{u}) = \text{bayes } (M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_{s'}(\vec{u'}) \text{ in}
\end{array}\right\}(5.7) \\
(s, P_s(\vec{u}))
\end{array}\right\}(5.5)
\end{array}\right\}(5.3) \\
\text{let } Bel_{new} = \text{bayes } \lambda(s, P_s(\vec{u}))\!:\!\_.\ (\text{expectation } (M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_s(\vec{u}))\ Bel \quad \}\ (5.4)
\end{array}
$$

Figure 5.13: Implementation of robotic mapping in PTP.

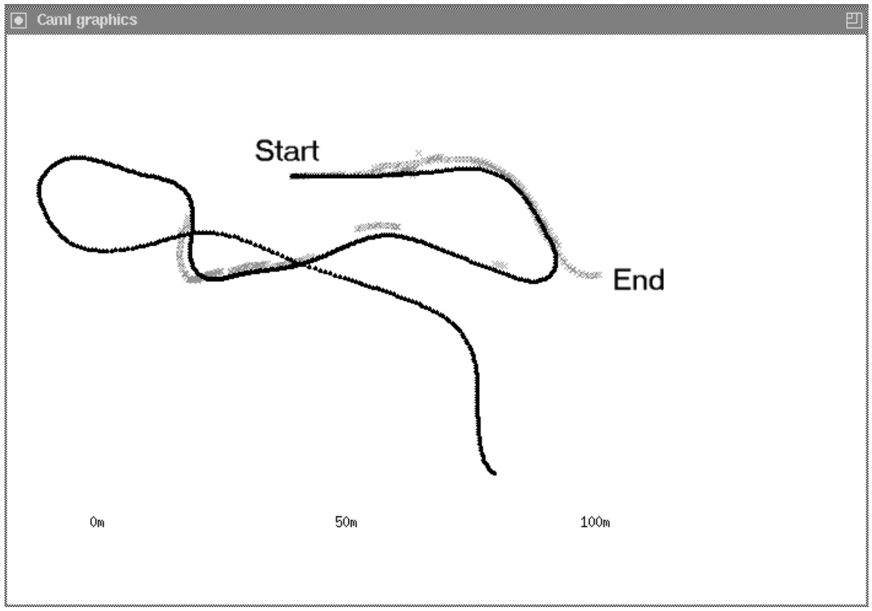


Figure 5.14: Raw odometry readings in the robotic mapping experiment.

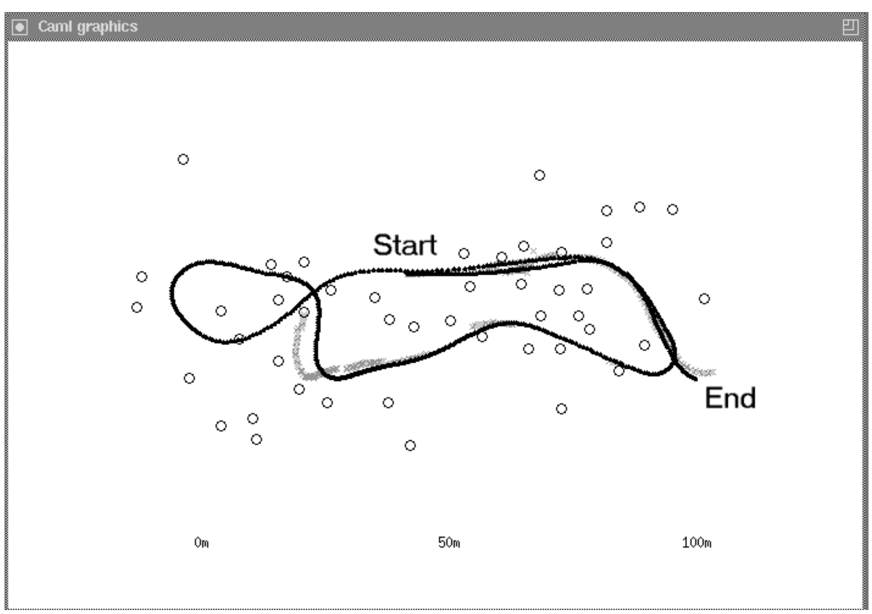


Figure 5.15: Result of the robotic mapping experiment.

Chapter 6

Conclusion

We have presented a probabilistic language PTP whose mathematical basis is sampling functions. PTP supports all kinds of probability distributions — discrete distributions, continuous distributions, and even those belonging to neither group — without drawing a syntactic or semantic distinction. We have developed a linguistic framework $\lambda_{\bigcirc}$ for PTP and demonstrated the use of PTP with three applications in robotics. To the best of our knowledge, PTP is the only probabilistic language with a formal semantics that has been applied to real problems involving continuous distributions. There are a few other probabilistic languages that are capable of simulating continuous distributions (by combining an infinite number of discrete distributions), but they require a special treatment such as the lazy evaluation strategy in [33, 59] and the limiting process in [24].

PTP does not support precise reasoning about probability distributions. Note, however, that this is not an inherent limitation of PTP due to its use of sampling functions as the mathematical basis; rather this is a necessary feature of PTP because precise reasoning about probability distributions is impossible in general. In other words, if PTP supported precise reasoning, it would support a smaller number of probability distributions and operations.

The utility of a probabilistic language depends on each problem to which it is applied. PTP is a good choice for those problems in which all kinds of probability distributions are used or precise reasoning is unnecessary. Robotics is a good example, since all kinds of probability distributions are used (even those probability distributions similar to $point\_uniform$ in Section 3.2 are used in modeling laser range finders) and also precise reasoning is unnecessary (sensor readings are inaccurate at any rate). On the other hand, PTP may not be the best choice for those problems involving only discrete distributions, since its rich expressiveness is not fully exploited and approximate reasoning may be too weak for discrete distributions.

Although we have presented only an operational semantics of PTP (which suffices for all practical purposes), a denotational semantics can also be used to argue that PTP is a probabilistic language. It may also answer important questions about PTP such as:

- What is exactly the expressive power of PTP?
- Can we encode any probability distribution in PTP?
- If not, what kinds of probability distributions are impossible to encode in PTP?

The challenge is that in the presence of fixed point constructs, measure theory does not come to our rescue because of recursive equations. Hence a domain-theoretic structure for probability distributions should be constructed to properly handle recursive equations. The work by Jones [30] suggests that such a structure could be constructed from a domain-theoretic model of real numbers [17].

The development of PTP is an effort to marry, in one of many possible ways, two seemingly unrelated disciplines: programming language theory and robotics. To programming language theory, it contributes a new linguistic framework $\lambda_{\bigcirc}$ and another installment in the series of probabilistic languages. To robotics, it sets a precedent that a high level formulation of a problem does not always have to be discarded when it comes to implementation. It remains to be seen in what other ways the two disciplines can be married.

Bibliography

[1] Experimental data. `http://www.acfr.usyd.edu.au/homepages/academic/enebot/dataset.htm`. Australian Centre for Field Robotics, The University of Sydney.

[2] Objective CAML. `http://caml.inria.fr`.

[3] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[4] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259. ACM Press, 2002.

[5] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–25. ACM Press, 2003.

[6] Z. M. Ariola and A. Sabry. Correctness of monadic state: an imperative call-by-need calculus. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, New York, NY, 1998.

[7] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, 1998.

[8] G. M. Bierman and V. de Paiva. Intuitionistic necessity revisited. Technical Report CSR-96-10, University of Birmingham, School of Computer Science, June 1996.

[9] G. Boudol. The recursive record semantics of objects revisited. *Lecture Notes in Computer Science*, 2028:269+, 2001.

[10] P. Bratley, B. Fox, and L. Schrage. *A guide to simulation*. Springer Verlag, 2nd edition, 1996.

[11] E. Charniak. *Statistical Language Learning*. MIT Press, Cambridge, Massachusetts, 1993.

[12] B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[13] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. Technical Report CMU-CS-03-164, School of Computer Science, Carnegie Mellon University, 2003.

[14] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society (Series B)*, 39(1):1–38, 1977.

[15] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer Verlag, New York, 2001.

[16] D. Dreyer. A type system for well-founded recursion. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2004.

[17] A. Edalat and P. J. Potts. A new representation for exact real numbers. In S. Brookes and M. Mislove, editors, *Electronic Notes in Theoretical Computer Science*, volume 6. Elsevier Science Publishers, 2000.

[18] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 174–185. ACM Press, 2000.

[19] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[20] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[21] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[22] M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes In Mathematics*, pages 68–85. Springer Verlag, 1981.

[23] T. G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.

[24] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 189–202. ACM Press, 1999.

[25] R. Harper, B. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[26] M. Henrion. Propagation of uncertainty in Bayesian networks by probabilistic logic sampling. In J. F. Lemmer and L. N. Kanal, editors, *Uncertainty in Artificial Intelligence 2*, pages 149–163. Elsevier/North-Holland, 1988.

[27] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, NY, 1980.

[28] A. H. Jazwinski. *Stochastic Processes and Filtering Theory*. Academic Press, New York, 1970.

[29] F. Jelinek. *Statistical Methods for Speech Recognition (Language, Speech, and Communication)*. MIT Press, Boston, MA, 1998.

[30] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[31] M. P. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[32] D. J. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Glasgow Functional Programming Workshop*, Glasgow, 1992. Springer Verlag.

[33] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 740–747. AAAI Press, 1997.

[34] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[35] S. A. Kripke. Semantic analysis of modal logic. I: Normal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[36] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 24–35. ACM Press, 1994.

[37] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[38] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349, 1997.

[39] C. Lüth and N. Ghani. Composing monads using coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 133–144. ACM Press, 2002.

[40] D. J. C. MacKay. Introduction to Monte Carlo methods. In M. I. Jordan, editor, *Learning in Graphical Models*, NATO Science Series, pages 175–204. Kluwer Academic Press, 1998.

[41] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM Press, 2003.

[42] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. Text of lectures originally given in 1983 and distributed in 1985.

[43] T. Mogensen. Roll: A language for specifying die-rolls. In V. Dahl and P. Wadler, editors, *5th International Symposium on Practical Aspects of Declarative Languages*, volume 2562 of *LNCS*, pages 145–159. Springer, 2002.

[44] E. Moggi. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, 1989.

[45] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[46] E. Moggi and A. Sabry. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming*, 11(6):591–627, Nov. 2001.

[47] E. Moggi and A. Sabry. An abstract monadic semantics for value recursion. *Theoretical Informatics and Applications*, 38(4):375–400, 2004.

[48] M. Montemerlo. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association*. PhD thesis, Robotics Institute, Carnegie Mellon University, 2003.

[49] M. Montemerlo, N. Roy, and S. Thrun. CARMEN: Carnegie Mellon Robot Navigation Toolkit. `http://www.cs.cmu.edu/~carmen/`.

[50] M. Montemerlo, W. Whittaker, and S. Thrun. Conditional particle filters for simultaneous mobile robot localization and people-tracking. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 695–701, Washington, DC, 2002. ICRA.

[51] A. Nanevski. From dynamic binding to state via modal possibility. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 207–218. ACM Press, 2003.

[52] A. Nanevski. A modal calculus for effect handling. Technical Report CMU-CS-03-149, School of Computer Science, Carnegie Mellon University, 2003.

[53] S. Park. A calculus for probabilistic languages. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 38–49. ACM Press, 2003.

[54] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 172–184. ACM Press, 2003.

[55] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[56] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.

[57] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 25–36. ACM Press, 1999.

[58] S. L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In C. A. R. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*. IOS Press, Amsterdam, 2001.

[59] A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 733–740. Morgan Kaufmann Publishers, Inc., 2001.

[60] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[61] D. Pless and G. Luger. Toward general analysis of recursive probability models. In J. Breese and D. Koller, editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-01)*, pages 429–436. Morgan Kaufmann Publishers, 2001.

[62] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.

[63] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, Feb. 1989.

[64] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165. ACM Press, 2002.

[65] W. Rudin. *Real and Complex Analysis*. McGraw-Hill, New York, 3 edition, 1986.

[66] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[67] A. Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.

[68] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997.

[69] N. Saheb-Djahromi. Probabilistic LCF. In J. Winkowski, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.

[70] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 8–17. ACM Press, 1999.

[71] A. K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, Department of Philosophy, University of Edinburgh, 1994.

[72] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

[73] S. Thrun. A programming language extension for probabilistic robot programming. In *Workshop notes of the IJCAI Workshop on Uncertainty in Robotics (RUR)*, 2000.

[74] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2000.

[75] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[76] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[77] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992.

[78] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4, 2003.

[79] G. Welch and G. Bishop. An introduction to the Kalman filter. Technical Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, 1995.