PλωNK:Functional Probabilistic NetKAT

ALEXANDER VANDENBROUCKE, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium

This work presents PλωNK, a functional probabilistic network programming language that extends Probabilistic NetKAT (PNK). Like PNK, it enables probabilistic modelling of network behaviour, by providing probabilistic choice and infinite iteration (to simulate looping network packets). Yet, unlike PNK, it also offers abstraction and higher-order functions to make programming much more convenient.

The formalisation of PλωNK is challenging for two reasons: Firstly, network programming induces multiple side effects (in particular, parallelism and probabilistic choice) which need to be carefully controlled in a functional setting. Our system uses an explicit syntax for thunks and sequencing which makes the interplay of these effects explicit. Secondly, measure theory, the standard domain for formalisations of (continuous) probablistic languages, does not admit higher-order functions. We address this by leveraging ω-Quasi Borel Spaces (ωQBSes), a recent advancement in the domain theory of probabilistic programming languages.

We believe that our work is not only useful for bringing abstraction to PNK, but that—as part of our contribution—we have developed the meta-theory for a probabilistic language that combines advanced features like higher-order functions, iteration and parallelism, which may inform similar meta-theoretic efforts.

Additional Key Words and Phrases: Probabilistic Programming, Network Modelling, Quasi-Borel Spaces, ω-QBS, NetKAT

1 INTRODUCTION

Probabilistic programming languages simplify the creation of probabilistic models. They separate the model from the algorithm that infers probabilities for it (e.g., Church [Goodman et al. 2012], Anglican [Wood et al. 2014], Gen [Cusumano-Towner et al. 2019], ProbLog [Fierens et al. 2015]). Instead of writing a custom procedure tailored to a particular model, the same generic algorithm is used for all programs written in the programming language. Thus, the algorithm can be re-used for many programs, lessening the implementation effort and maintenance burden of the probabilistic model.

In this work we develop a probabilistic programming language, called PλωNK.[1] PλωNK combines diverse features such as higher-order functions, probabilistic choice and parallelism. It is a domain specific language for probabilistically modelling computer networks. The main purpose of PλωNK is to model computer networks and network protocols at an abstract level, and verify a wide variety of properties of such models, e.g., latency, fault-tolerance, or the absence of routing loops. The motivation is the same as for formal verification of computer programs or the mechanical checking of proofs. Namely, during the design of complex networks or protocols, even the best designers are bound to make some mistakes or errors. A computer-checked specification can detect such deficiencies before they are deployed [Anderson et al. 2014; Foster et al. 2016].

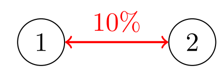
[1] Pronounced as “plonk”.

Authors' addresses: Alexander Vandenbroucke, KU Leuven, Celestijnenlaan 200 A, 3001, Leuven, Belgium, alexander.vandenbroucke@kuleuven.be; Tom Schrijvers, KU Leuven, Celestijnenlaan 200 A, 3001, Leuven, Belgium, tom.schrijvers@kuleuven.be.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

P$\lambda\omega$NK extends Probabilistic NetKAT (PNK) [Foster et al. 2016]. Here is a small PNK program:

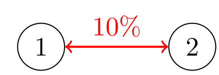
$$\big(\underbrace{sw = 1\,;\, drop \oplus_{0.1} sw \leftarrow 2}_{\text{node 1}} \,\&\, \underbrace{sw = 2\,;\, drop \oplus_{0.1} sw \leftarrow 1}_{\text{node 2}}\big)^*$$



The program on the left models the network on the right. This network consists of only two nodes, 1 and 2, with a bidirectional link between them. The link between the nodes is unreliable, e.g., it is a radio link with poor reception. This causes a 10% of the packets to be lost in transit.

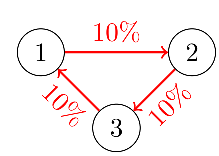
At this point it is not important to understand the meaning of this program exactly. Instead, note that this example already features a lot of repetition: the sub-programs to the left and to the right of the &-operator—modelling the behaviour of node 1 or 2, respectively—essentially mirror each other. Unfortunately, PNK offers no facilities to take advantage of this insight. The key advantage of P$\lambda\omega$NK over PNK is that we can exploit it by abstracting over the behaviour of both parts using functions:

$$\begin{array}{l} forward = \lambda src.\lambda dst.(sw = src;\, drop \oplus_{0.1} sw \leftarrow dst) \\ (\underbrace{forward\ 1\ 2}_{\text{node 1}} \,\&\, \underbrace{forward\ 2\ 1}_{\text{node 2}})^* \end{array}$$



The function *forward* captures the general forwarding behaviour of the nodes, independently of a particular node. While this change arguably does not make the program shorter, the improved readability and maintainability make writing and extending the program much more convenient. For instance, adding a third node is now much easier:

$$\begin{array}{l} forward = \lambda src.\lambda dst.(sw = src;\, drop \oplus_{0.1} sw \leftarrow dst) \\ (\underbrace{forward\ 1\ 2}_{\text{node 1}} \,\&\, \underbrace{forward\ 2\ 3}_{\text{node 2}} \,\&\, \underbrace{forward\ 3\ 1}_{\text{node 3}})^* \end{array}$$



Explicit Syntax for Thunks and Sequencing. Three distinct side-effects are in evidence in the above examples: (1) state—as we explain later, $sw \leftarrow 1$ modifies packets; (2) parallelism—the subprograms for node 1, 2 and 3 are run in parallel with &; and (3) probability—through the $\oplus_{0.1}$-operator. We carefully chose the previous example such that no arguments to the function contained any side-effects. Indeed, all arguments were constants, either 1, 2 or 3. Although this is already quite useful, we want to be more flexible in our full language, and also apply functions to non-constant expressions. In this case, should the side-effects of this expression be executed before the application and only the resulting value passed to the function (*Call-By-Value*)? Or, should the expression remain unevaluated, allowing the function to decide when to evaluate it (*Call-By-Name*)? Both strategies have their merits and there is no clear winner.

Rather than fix any particular order, we choose to explicitly segregate expressions into *computations* (which have side-effects) and *values* (which do not), loosely inspired by Call-By-Push-Value [Levy 2001].

The sequencing of side-effects then becomes explicit: either the computation is evaluated, producing a value which is then passed to the function, or the computation is explicitly turned into a value, by wrapping it in a *thunk*. While we use CBPV as an inspiration for the syntax and semantics, our language does not enjoy all theoretical properties of CBPV and thus does not model CBPV exactly.

Semantics. Our language, PλωNK, has higher-order functions, iteration and probabilistic choice. This significantly complicates the formalisation of the semantics of our language. After all, the standard approach to define denotational semantics for a continuous probabilistic language is based on measure theory. Yet, measure theory does not support general higher-order functions [Aumann et al. 1961], a central feature of our language. Thus, to support higher-order functions, we use (ω-)Quasi-Borel Spaces [Heunen et al. 2017; Vákár et al. 2019] as the domain of our denotational semantics. This is a recently developed alternative axiomatisation of probability theory, which admits higher-order functions. Moreover, it possesses the ωCPO structure required to model PNK style iteration (Kleene-star).

Another challenge we face is the interaction of higher-order functions, state, parallelism and probability. In our language, a computation can produce a function in a manner that is simultaneously probabilistic and non-deterministic (parallel), and also locally[2] modifying state. Moreover, we must avoid accidentally duplicating work of parallel branches, since parallel composition (&) is not idempotent (i.e., a program p is *not* equivalent to $p \,\&\, p$). As a result, defining the correct semantics for function application and sequencing is complicated and highly non-trivial. Note that this challenge is present in any calculus that supports function application, probability, parallelism and state—in fact, it is somewhat easier in our setting, as side-effects cannot occur in function arguments.

As we mentioned earlier, PλωNK's primary purpose is the *specification* of network models and the verification of properties of those models. It shares this purpose with PNK, which has several computational properties that make it well suited for this purpose: The denotational semantics of PNK can be approximated [Smolka et al. 2017b] computationally, through an iterative procedure. Moreover, at the cost of disallowing the *dup* operation, PNK has decidable program equivalence [Smolka et al. 2017a, 2019]. Thus, if we show that a small—hence, easy to prove correct—program is correct, we implicitly show that all equivalent larger programs are also correct.

We show that our language also possesses these properties, subject to some (minor) restrictions: the approximation procedure exists, if we forbid parallel choice between functions. Essentially, this restricts PλωNK's parallelism to the parallelism that is present in PNK. Also, we conjecture that disallowing the *dup* operation, as for PNK, results in decidable program equivalence for PλωNK. The specific contributions of this work are:

- We define the probabilistic programming language PλωNK, for modelling computer networks and protocols. It features higher-order functions, probabilistic choice and parallelism. PλωNK extends the earlier programming language PNK.
- PλωNK extends PNK with a simple type system. The type system is important, not only for rejecting invalid programs, but also to ensure that all programs *Strongly Normalise* [Pierce 2002]. On the one hand, strong normalisation indirectly makes our denotational semantics well-defined. On the other hand, we exploit this property for compiling PλωNK to PNK. Recall that PλωNK is a specification language, and the flexibility of general recursion and real arithmetic is not required.
- We define denotational semantics for PλωNK. As PλωNK contains higher-order functions, iteration and probabilistic choice, we need to leverage recent advances in the domain theory for probabilistic programs by Vákár et al. [2019]. They define an alternative formalisation of probability theory that admits higher-order functions, and iterations, the *ω-Quasi-Borel Spaces* (ωQBSes).

[2] By local, we mean that the state is not shared between different parallel or probabilistic branches.

We prove several well-definedness theorems about this denotational semantics. The (pen-and-paper) proofs for several of these theorems use logical relations whose definitions are interesting in their own right. We also prove that this semantics is a conservative extension of PNK's semantics as given by Smolka et al. [2017b]. The proofs themselves can be found in Appendix **??**.

- We develop a subclass of PλωNK programs which can be compiled into PNK. Through the type system, we restrict the parallelism in PλωNK to the parallelism present in PNK. This makes the values that are produced in parallel more predictable, allowing compilation to succeed. Moreover, PNK itself lies entirely within this class. We have mechanised the meta-theory of PλωNK and the compilation procedure with the aid of the Abella proof-assistant [Gacek 2008]. Theorems bearing a check mark (✓) have been mechanised. The proof scripts are available in the supplementary material.
  Compilation preserves the denotational semantics of PλωNK. Since our semantics is a conservative extension, the compiled PNK program behaves identically to the original PλωNK source program. These theorems are not easily encoded in Abella, and thus have been proven the classical way, with pen and paper (See Appendix **??**).
- We have implemented a prototype of PλωNK in Haskell. The prototype implements a small extension to PλωNK that performs type reconstruction. The implementation can be used to run the small examples that are presented in this article. It is part of the supplementary material.

2 OVERVIEW

2.1 A Brief Introduction to PNK

The central notion of PNK are *packet histories*, i.e., ordered sequences of packets. We write $\pi :: h$ for a history consisting of its most recent packet π, followed by the earlier history h. The empty history is written as $\langle\rangle$. The set of all packet histories is denoted PH.

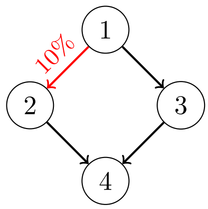
PNK programs operate on sets of these histories. Packets themselves are intended to be a simplified model of real-world binary network packets. As such, they consist of a number of header fields, which are assigned numeric values. Contrary to real-world packets, they do not contain a payload, because it is irrelevant for routing decisions. In our examples we commonly use the following headers: the switch the packet is currently at (sw) and the port the packet is currently at (pt).

PNK programs are constructed by composing a number of primitive operations. These primitives are predicates (e.g., *drop* or $sw = 1$), assignments (e.g., $pt \leftarrow 2$), and duplication. Recall that PNK programs operate on sets of packets histories. *Predicates* filter this set, allowing only specific histories. For instance, tests such as $sw = 1$ only allow histories where the first packet's header sw is set to 1, whereas *drop* denies all packets, producing an empty set. *Assignments* instead modify the histories in the set. For instance, the expression $pt \leftarrow 2$ sets the pt header to 2 for the first packet of every history. *Duplication dup* duplicates and prepends the first packet of every history in the set, i.e. for history $\pi :: h$, *dup* produces a history $\pi :: \pi :: h$.

Primitive operations can be composed in three ways:[3] sequentially, parallelly or probabilistically. *Sequential composition* (;) executes both operations one after the other, the output of the first becoming the input of the second. For instance, $sw = 1 \,; pt \leftarrow 2$ first filters out all histories where the sw header of the first packet is not 1, and then modifies the pt

[3] Actually, there is a distinction between composition for predicates and other operations, but it is not relevant here.

Network



PNK

Topology

$$
\begin{aligned}
t = \big(&sw = 1\,; pt = 2\,;\\
&(sw \leftarrow 2\,; pt \leftarrow 1) \oplus_{0.9} \mathit{drop}\big)\\
&\&\ (sw = 1\,; pt = 3\,; sw \leftarrow 3\,; pt \leftarrow 1)\\
&\&\ (sw = 2\,; pt = 4\,; sw \leftarrow 4\,; pt \leftarrow 2)\\
&\&\ (sw = 3\,; pt = 4\,; sw \leftarrow 4\,; pt \leftarrow 3)
\end{aligned}
$$

Routing

$$p = (sw = 1\,; pt \leftarrow 2)\,\&(sw = 2\,; pt \leftarrow 4)$$

PλωNK

Prelude

$$
\begin{aligned}
\mathit{send} &= \lambda src.\lambda dst.(sw = src\,; pt = dst)\\
\mathit{recv} &= \lambda src.\lambda dst.(sw \leftarrow dst\,; pt \leftarrow src)\\
\mathit{link} &= \lambda src.\lambda dst.(\mathit{send}\ src\ dst\,; \mathit{recv}\ src\ dst)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{forward} &= \lambda src.\lambda act.(sw = src\,; \mathit{force}\ act)\\
\mathit{to} &= \lambda dst.(pt \leftarrow dst)
\end{aligned}
$$

Topology

$$
\begin{aligned}
t = &(\mathit{send}\ 1\ 2\,;(\mathit{recv}\ 1\ 2 \oplus_{0.9} \mathit{drop}))\\
&\&\ \mathit{link}\ 1\ 3\,\&\ \mathit{link}\ 2\ 4\,\&\ \mathit{link}\ 3\ 4
\end{aligned}
$$

Routing

$$
\begin{aligned}
p &= \mathit{forward}\ 1\ (\mathit{thunk}\ (\mathit{to}\ 2))\\
&\quad \&\ \mathit{forward}\ 2\ (\mathit{thunk}\ (\mathit{to}\ 4))\\
p' &= \mathit{forward}\ 1\ \big(\underline{\mathit{thunk}\ (\mathit{to}\ 2 \oplus_{0.5} \mathit{to}\ 3)}\big)\\
&\quad \&\ \mathit{forward}\ 2\ \big(\mathit{thunk}\ (\mathit{to}\ 4)\big)\\
&\quad \&\ \underline{\mathit{forward}\ 3\ (\mathit{thunk}\ (\mathit{to}\ 4))}
\end{aligned}
$$

Main Expression

$$(p\,; t)^*\,; sw = 4$$

Fig. 1. PNK and PλωNK models of a small network consisting of 4 nodes.

header of the first packet of every history. *Parallel composition* (&) executes both operations independently and then takes the union of the resulting sets. For instance, $sw = 1\,\&\,sw = 2$ allows only packet histories that have the first packet's header set to either 1 or 2. Finally, *probabilistic choice* ($\oplus_r$) chooses either the left side with probability r, or the right with probability $1 - r$. For instance $sw \leftarrow 1 \oplus_{0.5} sw \leftarrow 2$, probabilistically chooses to set the first packet's header to 1 or to 2. If we sample from this expression, we see a set where the first packets' headers are either all set to 1, or all set to 2. The probability of either event is 50%.

2.2 Modelling in PNK

Let us consider how to model the network shown in the top-left corner of Figure 1 in PNK. The network consists of four nodes, with links from node 1 to nodes 2 and 3, and from nodes 2 and 3 to node 4. The objective is to send a network packet from node 1 to node 4, routed through either node 2 or node 3.

In order to accurately model the network, the program must model two independent aspects: the *network topology*, that is, the links connecting the nodes (and their behaviour), and the *routing programs* running on the nodes, receiving and forwarding incoming packets. The PNK program modelling the network is shown on the left of Figure 1.

Topology. The topology is captured by the term t on the left-hand side of Figure 1. It is a parallel composition of terms. It models, from top to bottom, the links from 1 to 2, from 1 to 3, from 2 to 4 and from 3 to 4. Each of the links consists of a number of statements, composed sequentially. For each link, it is verified that a packet is actually at the origin of the link by a series of guards (e.g., $sw = 1\,; pt = 3$, for the link from 1 to 3). Then, the packet is modified, setting *sw* and *pt* to the next switch on the link (e.g. $sw \leftarrow 3\,; pt \leftarrow 1$), thus transmitting the packet across the link. For the link between 1 and 2, we model unreliability by making a probabilistic choice ($\oplus_{0.9}$), choosing normal transmission 90% of the time and dropping (with *drop*) the packet 10% of the time. Packets that do not match the switch and port are rejected before their headers are modified.

Routing. The routing program is captured by the term p, a parallel composition of the routing programs for each node. For each node, it is verified that the *sw* field matches the node, and a node-specific routing program is then run: at node 1, packets are forwarded to node 2 and at node 2 packets are forwarded to node 4. Node 3 is unused for now.

Main Expression. The main expression (at the bottom of Figure 1) combines topology and routing, and provides an exit predicate. The $(\cdot)^*$, Kleene star, means iteration. Thus, the program $p\,; t$ is repeated until the exit predicate $sw = 4$ is satisfied. This predicate checks that the packet has arrived at node 4, its destination.

Having a PNK model of our network allows us to estimate the probability of certain queries, such as the probability that a packet reaches its destination (90% in this case) or measure the expected congestion. This is by no means an exhaustive list. A slightly modified program permits us to estimate the latency (i.e., the average length of a path), or by restricting the language, program equivalence becomes decidable, creating an easy way to verify correctness [Smolka et al. 2017a]. For additional examples and details, the interested reader should consult the work of Foster et al. [2016].

2.3 Extending PNK with functions

Even the small example from the previous section is quite tedious and repetitive to write. The root cause of this issue is the complete lack of abstraction facilities in PNK, since it is well-known that abstraction improves modularity and enables code re-use.

Arguably one of the most basic abstraction facilities available in programming languages is the venerable λ-abstraction. In P$\lambda\omega$NK, which additionally supports λ-abstractions, we instead encode the network topology (which features a lot of repetition) more concisely.[4] First we identify several recurring patterns and give them appropriate names. These are shown in the top-right corner of Figure 1. The primitive patterns are sending and receiving. which are combined to create a link. Functions such as *send*, *recv* and *link* could be defined in a library or a language prelude, to be reused by other programs.

We can now re-write the topology as show on the right of Figure 1. The links from 1 to 3, 2 to 4 and 3 to 4 simply call the appropriate function. The link from 1 to 2 must directly rely on sending and receiving, but even here, we can see the benefits of the approach in reducing duplication.

The functional re-write of the routing program p features a more advanced use of functions (program p on the right-hand side of Figure 1). The function *forward* takes a source node (*src*) and a forwarding action *act* to perform. This action is a *thunk*, a suspended computation, which can be executed or *forced*, with the primitive *force*. The justification for these constructions is explained in the next section. Since thunks are essentially functions,

[4] The code presented here is untyped, for didactic purposes. From Section 3 onwards we will use typed P$\lambda\omega$NK, although type reconstruction for P$\lambda\omega$NK is not difficult.

forward is a higher-order function. A more extensive use of higher-order functions can be found in the Gossip Protocols example provided with the supplementary material.[5]

To create the action *act* we use the function function *to*. This function sets the packet's *pt* header to the given destination *dst*.

In p, the forwarding program for node 1 calls *to*, immediately suspends the call (using *thunk*), and passes it to *forward*. Forwarding for node 2 proceeds in a similar fashion.

The main expression is as before. Due to the additional structure, the readability of the program has improved considerably. Furthermore, code re-use has gone up, making the program easier to change. For instance, to model an additional link, from 1 to 4, we only need to add the call (*link* 1 4), instead of the more lengthy ($sw = 1\,; pt = 4\,; sw \leftarrow 4\,; pt \leftarrow 1$).

For another example, suppose we want to change the forwarding behaviour of our network, such that node 1 now chooses to forward to either node 2 or node 3 with equal probability, then we need only extend p slightly, obtaining p'. The necessary additions have been highlighted in Figure 1. In short, this section demonstrates the advantage for readability and maintainability that PλωNK provides.

2.4 Explicit Thunks and Sequencing

As we have shown in the previous section, it is highly desirable that functions are higher-order, in the sense that functions—and PNK expressions—can occur as arguments to other functions (e.g., *forward*). Then it seems reasonable to expect to be able to write the following:

$$sw \leftarrow 0\,;(\lambda x.sw = 1)\ (sw \leftarrow 1)$$

However, this presents an issue, since $sw \leftarrow 1$ has a side-effect: it sets a header in the packet. The evaluation order is now important: if $sw \leftarrow 1$ is evaluated before the application, this program accepts the packet, otherwise it drops the packet. The former corresponds to a Call-By-Value (CBV) order, the latter to a Call-By-Name (CBN) order. There is no clear reason to prefer one over the other, and both are useful in practice.

Indeed, we decide to not fix any particular evaluation order. Instead, we segregate terms into values and computations, inspired by the syntax of Call-By-Push-Value (CBPV) [Levy 2001]. Computations can be evaluated (possibly with side-effects), as opposed to values, which cannot be directly evaluated. Functions (classified as computations themselves) can only be applied to values. PNK terms are also computations, so the expression above is invalid in PλωNK syntax, since $sw \leftarrow 1$ is not a value.

Instead, we obtain two possible variants, depending on whether CBV or CBN is intended (here the *to* is a primitive, *not* the function *to* defined earlier):

$$\begin{array}{ll} \textbf{CBV:} & sw \leftarrow 0\,;(sw \leftarrow 1)\ \mathit{to}\ y.((\lambda x.sw = 1)\ y) \\ \textbf{CBN:} & sw \leftarrow 0\,;(\lambda x.sw = 1)\ (\mathit{thunk}\ (sw \leftarrow 1)) \end{array}$$

In the first case, the primitive *to* (sequencing) evaluates $sw \leftarrow 1$, including side effects, and binds the value that is produced to the variable y, followed by applying the function to y. In the second case, $sw \leftarrow 1$ is thunked, and the function is applied to this thunk instead.

2.5 Semantics of Iteration

Foster et al. [2016] define the semantics of PNK in terms of measure theory. Semantically, a program denotes a function that maps sets of packet histories to a probability distribution over sets of packet histories. For example, the program $src = 1$, given input set A, returns

[5] https://bitbucket.org/AlexanderV/probnetkat-lambda/src/1d12d/gossip-protocols.pnk

a probability distribution that has probability 1 at the set $\{\pi :: h \in A \mid \pi.sw = 1\}$ (the notation $\pi.sw$ refers to the value of the sw header of π) and is zero everywhere else.

Iteration (*) is defined as an infinite stochastic process. The formalisation of this process is quite involved. Smolka et al. [2017b] give an equivalent, but much simpler definition, based on standard notions from domain theory.

In our work we retain this much simpler second definition, but extend it to support higher-order functions. However, since their domain is measure-theoretic, it does not support such functions. For this reason, we cannot use their domain directly. Instead, we rely on ωQBS, a domain developed by Vákár et al. [2019], which has domain-theoretic structure, supports measure-theory-like operations and admits higher-order functions.

2.6 Key Ideas

Denotational Semantics. We give a denotational semantics to our language within the ωQBS framework. In addition to PNK's semantics, ours also manages variable environments, and passes values (e.g. headers, constants, thunks) instead of just sets of packet histories.

The well-definedness of the semantics depends on two properties: First, to apply the domain-theoretic approach, we need to show that a specific notion of *continuity* holds. Second, we need a property that is similar to, or a consequence of, *strong normalisation*, but for denotational semantics. Informally, this property says that the denotation of a program produces only finitely many distinct values in a parallel fashion. The proofs for both properties have a similar structure: they proceed by induction on typing derivations of P$\lambda\omega$NK programs and make use of logical relations. The definitions of these logical relations are interesting in their own right (see Sections 5.3 and 5.4).

Approximation and Decidable Equivalence. The streamlined semantics of Smolka et al. [2017b] formalises an iterative approximation procedure for PNK programs. The idea is to expand the iterations (*) up to n times, for some finite n. We define a procedure to compile a P$\lambda\omega$NK program to PNK, while preserving the denotational semantics. Because our semantics is conservative with respect to the semantics of PNK, we can approximate the compiled program. Unfortunately, this compilation is only valid for a subclass of P$\lambda\omega$NK programs. Essentially, the trick is to impose additional restrictions in the type system, such that the parallelism in P$\lambda\omega$NK is limited to the parallelism that occurs in PNK.

Moreover, without the *dup* operation, PNK exhibits decidable program equivalence [Smolka et al. 2017a]. The *dup* operation duplicates the packet at the head of a packet history. Thus, removing this operation restrains all packet histories to the same length, making the state space of a program essentially discrete and finite. We conjecture that the same restriction also makes P$\lambda\omega$NK's state space discrete. However, under this restriction, PNK produces only discrete distributions. Many useful properties are expressible in this sub-language [Smolka et al. 2019]. However, it cannot express some relevant properties, e.g. latency. Hence, this setting is less interesting than the full language. For this reason, we focus on full P$\lambda\omega$NK. We revisit these issues in Section 6.

3 SYNTAX AND TYPE SYSTEM

3.1 Syntax of Terms

The syntax of P$\lambda\omega$NK (Figure 2) is a straightforward extension of the syntax of PNK with higher-order functions, thunks and sequencing. We segregate syntax terms into values and computations.

Terms.

$$
\begin{array}{llll}
x, y & \in & \mathit{Var} & \text{Variables} \\
h & \in & \mathit{Headers} & \text{Header names} \\
n & \in & \mathbb{N} & \text{Header values} \\
r & \in & \mathbb{R} & \text{Weights} \\
V & = & x \mid \mathit{unit} \mid h \mid n \mid \mathit{thunk}\ C & \text{Values} \\
P & = & \mathit{skip} \mid \mathit{drop} \mid V = V \mid \neg P \mid P \wedge P \mid P \vee P & \text{Predicates} \\
C & = & P \mid V \leftarrow V \mid \mathit{dup} \mid C ; C \mid C \,\&\, C \mid C \oplus_r C \mid C^* & \text{PNK computations} \\
 & \mid & \mathit{produce}\ V \mid \mathit{force}\ V \mid C\ \mathit{to}\ x.C \mid \lambda x{:}S.C \mid C\ V & \text{New computations}
\end{array}
$$

Types and Contexts.

$$
\begin{array}{ll}
S = \mathbf{1} \mid \mathcal{H} \mid \mathbb{N} \mid \mathcal{T}\,T & \text{Value types} \\
T = S \to T \mid \mathcal{P}\,S & \text{Computation types} \\
\Gamma = \emptyset \mid x{:}S, \Gamma & \text{Contexts}
\end{array}
$$

Fig. 2. PλωNK syntax.

Values V are either variables x, *unit* values, header names h (from a finite set *Headers*, e.g. *sw*, *pt*, ...), literals n (natural numbers, the values that can be assigned to a header) or *thunks* (suspended computations). Values are never evaluated, but thunks can be forced.

Computations C, on the other hand, can be evaluated, but cannot occur as the argument to a function. Consequently, only values can appear on the right-hand side of an application.

All original PNK constructs are computations C. *Predicates* P are a subsort of those. *Atomic* predicates are *skip*, *drop* or tests $V = V$. *Composite* predicates are negation $\neg$, conjunction ($\wedge$) or disjunction ($\vee$).[6] The remaining features inherited from PNK are (non-predicate) computations in PλωNK. They assign values to headers $V \leftarrow V$, duplicate packets with *dup*, compose sequentially (;) or in parallel (&), make a probabilistic choice $\oplus_r$ (we sometimes elide the weight r when it is not relevant) or iterate *. We call the computations that PλωNK inherits from PNK the *probabilistic* computations.

Finally, PλωNK adds the following computations to PNK: producing a value V (*produce* V), forcing a thunk (*force* V), sequencing computations with C_1 *to* $x.C_2$, defining a function $\lambda x{:}S.C$, or applying functions C to values V with ($C\ V$).

We also define *terminal* computations, i.e., computations that cannot be further evaluated:

$$
\begin{array}{ll}
R = & P \mid V \leftarrow V \mid \mathit{dup} \mid R ; R \mid R \,\&\, R \mid R \oplus_r R \mid R^* \\
 & \mid\ \mathit{produce}\ V \mid \lambda x{:}T.C
\end{array}
$$

3.2 Types and Type System

Because PNK terms cannot "go wrong" or get stuck, the language did not come with a type system. This is no longer true for PλωNK, which introduces stuck terms with the lambda calculus. For this reason, we enrich PλωNK with a simple type system.

The chief reason for choosing *simple* types is strong-normalisation (see Theorem 6.8). In order to not compromise PλωNK's suitability as a modelling language, certain properties,

[6] Contrary to previous work [Foster et al. 2016], we do distinguish the syntax for disjunctive and conjunctive predicates from parallel and sequential composition of computations, for improved clarity. Earlier developments used the same operators for both.

$$\boxed{\Gamma \vdash_v V : S}$$

$$\frac{x : S \in \Gamma}{\Gamma \vdash_v x : S} \qquad \frac{}{\Gamma \vdash_v \mathit{unit} : \mathbf{1}} \qquad \frac{}{\Gamma \vdash_v h : \mathcal{H}} \qquad \frac{}{\Gamma \vdash_v n : \mathbb{N}} \qquad \frac{\Gamma \vdash_c C : T}{\Gamma \vdash_v \mathit{thunk}\ C : \mathcal{T}\,T}$$

$$\boxed{\Gamma \vdash_c C : T}$$

$$\frac{}{\Gamma \vdash_c \mathit{skip} : \mathcal{P}\,\mathbf{1}} \qquad \frac{}{\Gamma \vdash_c \mathit{drop} : \mathcal{P}\,\mathbf{1}} \qquad \frac{\Gamma \vdash_v V_1 : \mathcal{H} \qquad \Gamma \vdash_v V_2 : \mathbb{N}}{\Gamma \vdash_c V_1 = V_2 : \mathcal{P}\,\mathbf{1}} \qquad \frac{\Gamma \vdash_c P : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c \neg P : \mathcal{P}\,\mathbf{1}}$$

$$\frac{\Gamma \vdash_c P_1 : \mathcal{P}\,\mathbf{1} \qquad \Gamma \vdash_c P_2 : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c P_1 \wedge P_2 : \mathcal{P}\,\mathbf{1}} \qquad \frac{\Gamma \vdash_c P_1 : \mathcal{P}\,\mathbf{1} \qquad \Gamma \vdash_c P_2 : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c P_1 \vee P_2 : \mathcal{P}\,\mathbf{1}}$$

$$\frac{\Gamma \vdash_v V_1 : \mathcal{H} \qquad \Gamma \vdash_v V_2 : \mathbb{N}}{\Gamma \vdash_c V_1 \leftarrow V_2 : \mathcal{P}\,\mathbf{1}} \qquad \frac{}{\Gamma \vdash_c \mathit{dup} : \mathcal{P}\,\mathbf{1}} \qquad \frac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C_1 \,;\, C_2 : T_2}$$

$$\frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 : T} \qquad \frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus C_2 : T} \qquad \frac{\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c C^* : \mathcal{P}\,\mathbf{1}}$$

$$\frac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \mathit{produce}\ V : \mathcal{P}\,S} \qquad \frac{\Gamma \vdash_v V : \mathcal{T}\,T}{\Gamma \vdash_c \mathit{force}\ V : T} \qquad \frac{\Gamma \vdash_c C_1 : \mathcal{P}\,S \qquad x : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1\ \mathit{to}\ x.C_2 : T}$$

$$\frac{x : S, \Gamma \vdash_c C : T}{\Gamma \vdash_c \lambda x : S.C : S \to T} \qquad \frac{\Gamma \vdash_c C : S \to T \qquad \Gamma \vdash_v V : S}{\Gamma \vdash_c C\ V : T}$$

Fig. 3. P$\lambda\omega$NK's typing rules.

such as approximation and program equivalence must remain decidable, requiring terminating reduction for all programs.

For simplicity, we elide other less essential features, such as sum and product types, but such features could be added without too much trouble.

The bottom part of Figure 2 shows the syntax of types. Types are divided into two kinds: value types and computation types. There are 4 forms of value types: unit types $\mathbf{1}$, header labels $\mathcal{H}$, header literals $\mathbb{N}$ and thunks $\mathcal{T}\,T$. Furthermore, there are 2 forms of computation types: function types $S \to T$ and producer types $\mathcal{P}\,S$. By construction, the argument position of a function type can only be a value type. Likewise, only value types can appear inside producer ($\mathcal{P}$) types. Since only these constructions bind variables, only value types appear in contexts.

Figure 3 shows the typing rules. We define two mutually recursive typing judgements: $\Gamma \vdash_v V : S$ for typing value terms, and $\Gamma \vdash_c C : T$ for typing computation terms.

The rules for values are straightforward. Predicates are typed as computations, and always have type $\mathcal{P}\,\mathbf{1}$ because they do not produce useful results, only useful side-effects. This also applies to the rules for modification and duplication (see below).

When sequentially composing computations $C_1 \,;\, C_2$, only C_2 determines the type of the whole computation. For parallel composition and choice, the types of both computations must be *the same*. Intuitively, the former discards the value produced by the first computation, while the latter two must somehow combine the two produced values, requiring them to have the same type.

Iteration C^* has the same type as C, which is only allowed to be $\mathcal{P}\,\mathbf{1}$, for the following reason. Consider that the iteration could be zero or more times, and thus in the zero case would require inventing a value of an arbitrary type, which we cannot do unless we restrict iteration to a fixed type with a known value—the unit type. The remaining rules in Figure 3 are derived from CBPV [Levy 2001].

However, unlike CBPV, PλωNK does *not* exhibit certain type isomorphisms, for instance:

$$S \to S' \to T \ncong S' \to S \to T$$

This is because in our language computations of function type can have side-effects without being applied, whereas in CBPV computation of function type are only evaluated upon application. In particular, applying a computation of type $S \to S' \to T$ to a value x produces a computation of type $S' \to T$ which may have side-effects that depend on x.

In the previous sections, we have also used "top-level" definitions of the form $f = \cdots$. These are to be understood as syntactic sugar. They desugar into λ-abstraction and application as follows:

$$\boxed{\begin{array}{l} f_1 = C_1 \\ C_2 \end{array}} \rightsquigarrow (\lambda y\!:\!\mathcal{T}\,T.[f_1 \mapsto \mathit{force}\ y]C_2)\ (\mathit{thunk}\ C_1)$$

where y is fresh, and $\vdash_c C_1\!:\!T$. Recall that only values may be bound to variables. Since the C_1 on the right-hand side of the equality is a computation, we first convert this expressions to a value by thunking them, and then forcing them where they occur in C_2.

4 A CONVENIENT CATEGORY FOR PλωNK

Measure theory is the usual model for continuous[7] distributions. However, for our case classical measure theory has a critical shortcoming: function spaces of measure spaces are not necessarily measurable themselves, making measure theory unsuitable as the model of a programming language with λ-abstraction [Aumann et al. 1961]. Put in a different way, the category of measurable spaces is not Cartesian closed.

Quasi-Borel Spaces [Heunen et al. 2017] are a recent advancement in the state-of-the-art of the semantics of probabilistic programs, which *are* Cartesian closed and provide an alternative formalisation of probabilistic structures. An even more recent development are the ω-Quasi-Borel Spaces [Vákár et al. 2019], which additionally provide ωCPO structure. We require this structure to model iterations.

The remainder of this section provides a brief overview of ω-Quasi-Borel Spaces, which we use as the semantic domain for the denotational semantics of PλωNK that is presented in the next section. Eager readers may skip ahead to Section 5 on a first reading and come back when they want more detail.

4.1 ω-Complete Partial Orders

Semantically modelling the behaviour of PλωNK or indeed plain PNK requires a semantic domain that captures the recursive nature of iterations C^*. For this purpose we use a partially ordered domain and make sure that the denotation increases with each iteration

[7] At first glance, it might seem counter-intuitive that PλωNK could admit continuous probability distributions. However, Foster et al. [Foster et al. 2016] show an example of a PNK program where this is the case: let p be a program that outputs two distinct packets with equal probability, then $p\,;(\mathit{dup}\,;p)^*$ denotes a continuous distribution.

step, i.e., it ascends. The result of the entire iteration is then given by the least upper bound of the denotations of the iteration steps, and we must choose the ordering in such a way that these least upper bounds always exist. The ω-Complete Partial Orders (ωCPOs) are the orders with this property: in an ωCPO the least upper bounds of ascending chains always exist. Let us now define these concepts more precisely.

Definition 4.1. Let $\langle P, \sqsubseteq \rangle$ be some poset, an *ω-chain* is a sequence $(x_n)_{n\in\mathbb{N}}$ for $x_n \in P$, such that $\forall i, j \in \mathbb{N} : i \leq j$ implies $x_i \sqsubseteq x_j$. We will sometimes write such a chain as $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$.

Definition 4.2. The poset P is an *ω-Complete Partial Order (ωCPO)* when every ω-chain has a least upper bound (lub) $\bigsqcup_{n\geq 0} x_n \in P$ (sometimes also denoted $\bigvee_{n\geq 0} x_n$). The *least upper bound* is the smallest element of P that is larger than every x_n. More formally,

$$\forall n \in \mathbb{N} : x_n \sqsubseteq \bigsqcup_{n\geq 0} x_n, \text{ and } \forall z \in P : (\forall n \in \mathbb{N} : x_n \subseteq z) \Rightarrow \bigsqcup_{n\geq 0} x_n \sqsubseteq z.$$

Example 4.3. The powerset 2^X of any set X is an ωCPO when ordered by subset inclusion ($\subseteq$). The lub of any ω-chain $X_0 \subseteq X_1 \subseteq \cdots$ (with $X_i \subseteq X$) is precisely the union $\bigcup_{i\geq 0} X_i$. In fact, $\langle 2^X, \subseteq \rangle$ is a complete lattice, meaning that any subset of 2^X has a lub.

Example 4.4. The functions $f : X \to P$ into an ωCPO $\langle P, \sqsubseteq_P \rangle$ also form an ωCPO, under the pointwise order $\preceq$, defined as $f \preceq g \iff \forall x \in X : f(x) \sqsubseteq_P g(x)$. In this instance, we usually denote the lub by $\bigvee$.

Continuous functions between two ωCPOs $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ are monotone functions $f : P \to Q$ such that f preserves least upper bounds, i.e. for all ω-chains $(a_n)_{n\in\mathbb{N}}$ in P,

$$f(\bigsqcup_{n\geq 0} x_n) = \bigsqcup_{n\geq 0} f(x_n).$$

Note that the first lub is in P, the second in Q. The monotonicity requirement ensures that this second lub actually exists, by ensuring that $(f(x_n))_{n\in\mathbb{N}}$ is an ω-chain.

The ωCPOs and continuous functions between them form a Cartesian closed category, i.e:

- the composition of two continuous functions is continuous;
- ωCPOs are closed under Cartesian products, i.e., given two ωCPOs P and Q, $P \times Q$ is also an ωCPO; and
- continuous functions between two ωCPOs also form an ωCPO. The order is the pointwise order.

4.2 Quasi-Borel Spaces

In P$\lambda\omega$NK we give semantics to higher-order functions and probabilistic choice. For this reason we need a semantic domain which is both Cartesian closed and admits a probabilistic power domain. As we mentioned previously, we cannot rely on measure theory, the usual choice for probabilistic domains, since it is not Cartesian closed. Instead, we use Quasi-Borel Spaces(QBS) [Heunen et al. 2017], which are Cartesian closed.

However, in order to define QBSes, we must revisit some basic definitions from measure theory first. We limit the treatment of measure theory to the essentials needed to understand Quasi-Borel Spaces. For instance, we do not discuss general measure spaces, but constrain ourselves to a particular set of measurable sets on the reals, the Borel sets:

Definition 4.5. The Borel sets $\mathfrak{B}$ are the least collection of subsets of $\mathbb{R}$, such that:

- the intervals $[a, b]$ are Borel sets for $a, b \in \mathbb{R}$,
- the complement of a Borel set is a Borel set, and
- countable unions of Borel sets are Borel sets.

A probability measure is a function $\mu : \mathfrak{B} \to [0, 1]$ satisfying $\mu(\mathbb{R}) = 1$ and $\mu(\bigcup S_n) = \sum \mu(U_n)$, for any countable sequence of disjoint Borel sets $(S_n)_{n\in\mathbb{N}}$. A function $f : \mathbb{R} \to \mathbb{R}$ is called *measurable* if its inverse image maps Borel sets to Borel sets. Symbolically, for all $B \in \mathfrak{B}$:

$$f^{-1}(B) = \{x \in \mathbb{R} \mid f(x) \in B\} \in \mathfrak{B}.$$

Such functions can be integrated with respect to a measure. For a *non-negative* real-valued measurable function $f : \mathbb{R} \to \mathbb{R}$, the integral of f with respect to a probability measure μ is defined as:

$$\int f \, d\mu = \sup_{(U_n)} \sum_n \left(\mu(U_n) \inf_{x \in U_n} f(x) \right)$$

where (U_n) ranges over finite partitionings of $\mathbb{R}$ into Borel sets. If f is allowed to be negative, its integral is:

$$\int f \, d\mu = \int f^+ \, d\mu - \int f^- \, d\mu$$

where $f^+ = \max(f, 0)$, $f^- = \max(-f, 0)$.

Measure theory generalises Borel sets to the measurable subsets of a measurable space X, and generalises measurable functions on $\mathbb{R}$ to measurable functions between measurable spaces, whose inverse image always maps measurable sets to measurable sets.

Quasi-Borel Spaces [Heunen et al. 2017] are an alternative to measure theory, starting from the from the notion of a random element $\mathbb{R} \to X$ instead of measurable sets.

Definition 4.6. A Quasi-Borel Space (QBS) $\langle X, M_X \rangle$ is a set X together with a set of functions M_X such that the following conditions are met:

- If $\alpha : \mathbb{R} \to X$ is constant, then $\alpha \in M_X$;
- if $\alpha \in M_X$, and $f : \mathbb{R} \to \mathbb{R}$ is measurable, then $\alpha \circ f \in M_X$;
- let $\mathbb{R} = \bigcup_{n\in\mathbb{N}} U_n$ where the U_n are pairwise disjoint Borel sets, if $\alpha_n \in M_X$ for all $n \in \mathbb{N}$, then $\beta \in M_X$ where $\beta(r) = \alpha_n(r)$ if $r \in U_n$.

Essentially, M_X must contain all constant functions, and must be closed under pre-composition with a measurable function or countable case-splitting. A function $f : X \to Y$ is a morphism from $\langle X, M_X \rangle$ to $\langle Y, M_Y \rangle$ if for all $\alpha \in M_X$, $f \circ \alpha \in M_Y$.

There are two canonical ways to turn a set X into a QBS. One option is to simply include *all* functions $\mathbb{R} \to X$ in M_X. Another option is to take as random elements all measurably piece-wise constant functions, i.e., those functions from $\mathbb{R}$ to X that are piece-wise constant on measurable sets of $\mathbb{R}$.

The quasi-Borel spaces together with their morphisms form a category. Moreover, this category admits products, co-products and function spaces. That is, unlike the category formed by measurable spaces and measurable functions between them, this category is Cartesian closed [Heunen et al. 2017, Proposition 18].

4.3 ω-Quasi-Borel Spaces

Definition 4.7. An ω-Quasi-Borel Space is a triple $\langle X, M_X, \sqsubseteq_X \rangle$ such that:

- $\langle X, M_X \rangle$ is a quasi-Borel Space,
- $\langle X, \sqsubseteq_X \rangle$ is an ωCPO, and

- $\left(\bigvee_{n\geq 0}\alpha_n\right) \in M_X$ for all ω-chains $\alpha_0 \preceq \alpha_1 \preceq \cdots$ (with $\alpha_n \in M_X$), where $\preceq$ is the point-wise order on M_X.

Informally, an ωQBS is a QBS that is also an ωCPO, and whose random elements are closed under pointwise lubs of ω-chains.

The morphisms between ωQBSes are those morphisms between the underlying ωQBSes, which are also Scott-continuous between the underlying ωCPOs.

We reiterate two examples from Vákár et al. [2019]:

Example 4.8. Real values have the ωQBS, $\langle \mathbb{R}, M_{\mathbb{R}}, =\rangle$, with $M_{\mathbb{R}}$ the set of measurable functions from $\mathbb{R}$ to $\mathbb{R}$. Alternatively, consider $\mathbb{W} = \langle [0,\infty], M_{\mathbb{W}}, <\rangle$ the space of weights, where $M_{\mathbb{W}}$ is the set of measurable functions from $\mathbb{R}$ to $[0,\infty]$.

Just like QBS, ωQBS forms a category that is closed under products, co-products and exponentials (functions). This category is Cartesian closed. With ωQBSes we finally have a mathematical concept which unifies iteration, probabilistic choice and higher-order functions.

4.4 A Commutative Probabilistic Powerdomain

The denotational semantics we give to P$\lambda\omega$NK is a monadic semantics: it allows the structuring of the semantics in a compositional fashion [Moggi 1991]. This section explains how to define a monad suitable for expressing probabilistic computations as an ωQBS.

The idea, as explained by Vákár et al. [2019], is to treat distributions as expectation or integration operators. The distribution monad $D(X)$ is (a submonad[8] of) the continuation monad $C(X) = (X \to \mathbb{W}) \to \mathbb{W}$, where the arrows ($\to$) denote ωQBS morphisms. The idea is that for a $\mu \in C(X)$ and $f : X \to \mathbb{W}$, $\mu(f)$ computes the integral of f with respect to μ, in more traditional notation, $\mu(f) = \int f \, d\mu$.

The unit (*return*) and composition ($\ggg$) of this monad are given by:

$$\mathit{return}\ x = \lambda k \to k\ x \qquad\qquad m \ggg f = \lambda k \to m\ (\lambda x \to f\ x\ k)$$

In more traditional language these represent integrating with δ_x (Dirac-delta of x) and the integral $\int_x \int k\ d(f\ x)\ dm$, respectively. The details of this construction are beyond the scope of this article. The most important results are:

- This construction satisfies the requirements of synthetic measure theory, implying that the results of measure theory continue to hold in this new setting. In particular, we can do addition and scalar multiplication of distributions.
- The monad consists of the *s-finite measures and kernels*, meaning that the monad is commutative: operations can be re-ordered [Staton 2017].

5 DENOTATIONAL SEMANTICS

A denotational semantics maps terms (values and computations) into a semantic domain. A term's domain is determined by its type, therefore, we must foremost discuss the semantics of types.

5.1 Semantics of Types

Figure 4 shows the interpretations of the different types. Each type denotes an ωQBS, which we denote by just the underlying set X instead of the full triple $\langle X, M_X, \sqsubseteq_X\rangle$.

[8] Specifically, it is the smallest ωQBS that is a full sub-ωCPO of C(X) and contains the randomisable random elements.

$$
\begin{aligned}
[\![\mathbf{1}]\!] &= \langle\{()\}, M_{()}, (=)\rangle \\
[\![\mathcal{H}]\!] &= \langle \mathit{Headers}, M_{\mathit{Headers}}, (=)\rangle \\
[\![\mathbb{N}]\!] &= \langle \mathbb{N}, M_{\mathbb{N}}, (=)\rangle \\
[\![\mathcal{T}\,T]\!] &= [\![T]\!] \\
[\![S \to T]\!] &= 2^{PH} \to D([\![T]\!]^{[\![S]\!]} \rightharpoonup 2^{PH}) \\
[\![\mathcal{P}\,S]\!] &= 2^{PH} \to D([\![S]\!] \rightharpoonup 2^{PH}) \\
A \rightharpoonup B &= A \to B_\perp
\end{aligned}
$$

Fig. 4. Denotations of types and contexts.

Unit types $\mathbf{1}$ are denoted by nullary products, whose only inhabitant is written as $()$. Header types $\mathcal{H}$ are denoted by the finite set of header labels and header literals $\mathbb{N}$ are denoted by the natural numbers. In these three cases, the underlying ωCPO is given by the discrete order $(=)$[9] and the underlying ωQBS can be created using either of the canonical methods described in Section 4.2. Thunk types simply denote the denotation of the thunked computation type.

Computation types need to capture the side effects: state, parallelism, and probabilistic choice. The denotations of both sorts of computation types are of a similar form:

$$
\overbrace{2^{PH}}^{\text{input state}} \to D(\underbrace{X \rightharpoonup \overbrace{2^{PH}}^{\text{output state}}}_{\text{parallel value and state}})
$$

where D is the distribution monad defined in Section 4.4, the arrow $(\to)$ denotes ωQBS-morphism and the harpoon $(\rightharpoonup)$ denotes partial maps. Partial maps $f : A \rightharpoonup B$ are equivalent to total functions $\widehat{f} : A \to B_\perp$, where $B_\perp$ is B extended with a distinguished element, $\perp_B$, preceding all other elements. For any $a \in A$, $\widehat{f}(a) = \perp_B$ then means that f is undefined on a. Hence, the domain $\mathit{dom}(f)$ of a partial map f is defined as $\mathit{dom}(f) = \{x \mid \widehat{f}(x) \neq \perp\}$. The partial maps are not required to be continuous (but they do form an ωCPO and an ωQBS).

Note that we use the power set of packet histories to represent the state. The ωQBS of this powerset is $\langle 2^{PH}, M_{2^{PH}}, \subseteq\rangle$ where $M_{2^{PH}}$ are the random elements of the exponential 2^{PH} of $\langle 2, M_2, =\rangle$ and $\langle PH, M_{PH}, =\rangle$.

We use partial maps $X \rightharpoonup 2^{PH}$ to model parallelism in both the value (X) and the state (2^{PH}). The idea is that a particular value comes with a particular state. Since the map is partial, not all values are necessarily present. Note that we cannot use total maps and the empty set to indicate the absence of a value, since the empty set is a *valid* state.

For producer types $\mathcal{P}\,S$, $X = [\![S]\!]$. For function types $S \to T$, $X = [\![T]\!]^{[\![S]\!]}$, i.e., the morphisms (functions) from $[\![S]\!]$ to $[\![T]\!]$.

Finally, the semantics $[\![S]\!]$ extends naturally to an interpretation on contexts (recall that contexts only contain value types):

$$
[\![x_1 : S_1, \ldots, x_k : S_k]\!] = [\![S_1]\!] \times \cdots \times [\![S_k]\!]
$$

[9] In this order, everything is incomparable to everything, unless they are the same element.

That is, contexts are denoted by the product of the denotations of the types within the context. For convenience, however, for an environment $\rho \in [\![\Gamma]\!]$, we will write $\rho(x)$ to look up a variable x, and $[x \mapsto v]\rho$ to update the value of x in ρ.

5.2 Semantics of Terms

Next, we define denotational semantics for terms (Figure 5). The semantics are divided into three (mutually recursive) categories: semantics for values, for predicates and for other computations. All semantic functions take an environment $\rho \in [\![\Gamma]\!]$ as their first argument. Predicates and computations also take a set of packet histories $A \in 2^{PH}$.

5.2.1 Values. For values, the semantics is relatively straightforward: variables can be looked up in the environment ρ. The *unit*s, headers and literals map to their corresponding constants. A thunk *thunk* C partially applies the denotation of C to the current environment ρ. Recall that $[\![C]\!]\ \rho$ is a function that expects a set of packet histories. When the thunk is forced this function is applied to the current state. To see why this makes sense, consider that it is the location where the thunk is created that determines the scope of the *variables* in C, but it is the location where it is forced which determines its *state*.

5.2.2 Predicates. Predicates filter sets of packet histories, that is, they allow or reject particular packet histories.

Predicates *skip* and *drop* allow, respectively disallow, all packet histories. A guard $V_1 = V_2$ only allows packet histories where the first packet's header $[\![V_1]\!]\ \rho$ has the specified value $[\![V_2]\!]\ \rho$ (accessing header f of a packet π is written as $\pi.f$). Negation ($\neg$) only retains packets that are dropped by its argument. Finally, conjunction ($\wedge$) and disjunction ($\vee$) take the intersection and union respectively.

5.2.3 Computations. The semantics for computations returns values in the monad D. Since D is a monad, we use the well-understood *do-notation* to construct monadic expressions, de-sugaring straightforwardly to monadic bind ($\gg\!=$).

The first three cases are non-probabilistic. Moreover, they all produce the unit value (), they return a map from () to a modified set of histories.[10] Predicates P filter packet histories according to the predicate semantics $[\![P]\!]^p\ \rho\ A$. Modifications change the first packet's header for every packet history in the input set (setting a packet π's header f to x is written $\pi[f \mapsto x]$). Similarly, *dup* duplicates the first packet.

Sequential composition ($C_1\,; C_2$) first evaluates C_1, and then C_2 (returning the value of C_2). Since the evaluation of C_2 does not depend on the value of C_1, all packet histories of C_1 are simply aggregated and passed to C_2.

Parallel composition $C_1 \,\&\, C_2$ combines the results (values and states) of both computations. Since the results of C_1 and C_2 are captured by the partial maps μ_1 and μ_2, it must combine these maps. The pointwise lub $\mu_1 \bigvee \mu_2$ is exactly what is needed: the domain of the lub is the union of the domains of μ_1 and μ_2, so it has all the values of both computations, and the sets of packet histories it maps a value x onto is the union of $\mu_1(x)$ and $\mu_2(x)$.

Choice $C_1 \oplus_p C_2$, reweighs both C_1 and C_2, by p and $1-p$ respectively, and adds the resulting distributions. Note that the sum is a probability distribution, i.e. all weights sum to one. Since D satisfies the requirements of synthetic measure theory, scalar multiplication and addition behave as one would intuitively expect, multiplying and adding probabilities.

[10] By convention, λ with an arrow ($\rightarrow$) is abstraction in the meta-language (morphisms in ωQBS) and λ with dot . refers to abstraction in the object language.

Values $[\![V]\!] : [\![\Gamma]\!] \to [\![S]\!]$ for $\Gamma \vdash_v V : S$

$$
\begin{aligned}
[\![x]\!]\ \rho &= \rho(x) \\
[\![unit]\!]\ \rho &= () \\
[\![h_i]\!]\ \rho &= h_i \\
[\![i]\!]\ \rho &= i \\
[\![thunk\ C]\!]\ \rho &= [\![C]\!]\ \rho
\end{aligned}
$$

Predicates $[\![P]\!]^p : [\![\Gamma]\!] \to 2^{PH} \to 2^{PH}$ for $\Gamma \vdash_c P : \mathcal{P}\,\mathbf{1}$

$$
\begin{aligned}
[\![skip]\!]^p\ \rho\ A &= A \\
[\![drop]\!]^p\ \rho\ A &= \emptyset \\
[\![V_1 = V_2]\!]^p\ \rho\ A &= \{\pi :: h \in A \mid \pi.([\![V_1]\!]\ \rho) = [\![V_2]\!]\ \rho\} \\
[\![\neg P]\!]^p\ \rho\ A &= A - [\![P]\!]^p\ \rho\ A \\
[\![P_1 \wedge P_2]\!]^p\ \rho\ A &= [\![P_1]\!]^p\ \rho\ A \cap [\![P_2]\!]^p\ \rho\ A \\
[\![P_1 \vee P_2]\!]^p\ \rho\ A &= [\![P_1]\!]^p\ \rho\ A \cup [\![P_2]\!]^p\ \rho\ A
\end{aligned}
$$

Computations $[\![C]\!] : [\![\Gamma]\!] \to [\![T]\!]$ for $\Gamma \vdash_c C : T$

$$
\begin{aligned}
[\![P]\!]\ \rho\ A &= return\ (\lambda() \to [\![P]\!]^p\ \rho\ A) \\
[\![V_1 \leftarrow V_2]\!]\ \rho\ A &= \ \mathbf{let}\ f = [\![V_1]\!]\ \rho, j = [\![V_2]\!]\ \rho \\
&\quad\ \mathbf{in}\ return\ (\lambda() \to \{\pi[f \mapsto j] :: h \mid \pi :: h \in A\}) \\
[\![dup]\!]\ \rho\ A &= return\ (\lambda() \to \{\pi :: \pi :: h \mid \pi :: h \in A\}) \\
[\![C_1\,;C_2]\!]\ \rho\ A &= \mathbf{do}\ \ \mu \leftarrow [\![C_1]\!]\ \rho\ A \\
&\qquad\quad [\![C_2]\!]\ \rho\ \left(\textstyle\bigcup_{x \in dom(\mu)} \mu(x)\right) \\
[\![C_1\,\&\,C_2]\!]\ \rho\ A &= \mathbf{do}\ \ \mu_1 \leftarrow [\![C_1]\!]\ \rho\ A \\
&\qquad\quad \mu_2 \leftarrow [\![C_2]\!]\ \rho\ A \\
&\qquad\quad return\ (\mu_1 \textstyle\bigvee \mu_2) \\
[\![C_1 \oplus_r C_2]\!]\ \rho\ A &= r([\![C_1]\!]\ \rho\ A) + (1-r)([\![C_2]\!]\ \rho\ A) \\
[\![C^*]\!]\ \rho\ A &= \ \textstyle\bigsqcup_{n \geq 0} ([\![C^n]\!]\ \rho\ A)\ \ \mathbf{where} \\
&\quad C^0 \qquad = skip, \\
&\quad C^{n+1} \quad = skip\,\&(C\,; C^i), n \geq 0 \\
[\![produce\ V]\!]\ \rho\ A &= return\ (\lambda([\![V]\!]\ \rho) \to A) \\
[\![force\ V]\!]\ \rho\ A &= [\![V]\!]\ \rho\ A \\
[\![\lambda x : S.C]\!]\ \rho\ A &= return\ (\lambda(\lambda v \to [\![C]\!]\ ([x \mapsto v]\rho)\ ) \to A) \\
[\![C\ V]\!]\ \rho\ A &= \mathbf{do}\ \ \mu \leftarrow [\![C]\!]\ \rho\ A \\
&\qquad\quad \Xi\{f\ ([\![V]\!]\ \rho)\ \mu(f) \mid f \in dom(\mu)\} \\
[\![C_1\ \ to\ x.C_2]\!]\ \rho\ A &= \mathbf{do}\ \ \mu \leftarrow [\![C_1]\!]\ \rho\ A \\
&\qquad\quad \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\}
\end{aligned}
$$

$$
\mathbf{where}\ \Xi\{m_1, \ldots, m_k\} = \mathbf{do}\ \
\begin{aligned}
&\mu_1 \leftarrow m_1 \\
&\vdots \\
&\mu_k \leftarrow m_k \\
&return\ (\mu_1 \textstyle\bigvee \cdots \bigvee \mu_k)
\end{aligned}
$$

Note: $X \to Y$ means ωQBS-morphisms from X to Y.

Fig. 5. Denotations of terms.

Iteration (Kleene-star C^*) is defined as it is for PNK, using least fixed-point semantics. It can be understood as the least fixed-point of the function $\big(\gg\!\!= \lambda\mu \to [\![\mathit{skip}\,\&\,C]\!]\ \rho\ \mu(())\big)$. Or, operationally, we simply take the least upper bound of iterating C for $0, 1, \ldots$ times. This definition is identical to Smolka et al.'s [2017b], although on a different ωCPO. Treating probabilistic loops as fixed-points goes at least as far back as Kozen's [1981] early work on probabilistic program semantics.

To produce (*produce* V) a value V, we return the map from $[\![V]\!]\ \rho$ to the current set of packet histories A. Here we use the notation $\lambda([\![V]\!]\ \rho) \to A$ to define the map that is A on $[\![V]\!]\ \rho$ and $\perp$ everywhere else.

Thunks are forced by evaluating their semantics, and then applying the result of that semantics to the input set of packet histories. Recall that the semantics of a thunk corresponds to a partially applied semantics of the thunked computation.

For an abstraction $\lambda x : S.C$ we construct a map with a domain containing only one particular anonymous function in the meta-language. This function takes a $v \in [\![S]\!]$, extends the environment ρ with v, and runs $[\![C]\!]$ in this new environment.

For application, we must first sample from the computation C. This produces a map from $[\![S]\!] \to [\![T]\!]$ to sets of packet histories. Each *unique* function f in the map μ is then applied to the argument $[\![V]\!]\ \rho$ and the corresponding set of packet histories $\mu(f)$. Every unique function f corresponds to one or more parallel branches that produce this value, $\mu(f)$ aggregates the states of each of those branches. This produces a set of distributions, which we can collapse using the Ξ operator (also defined in Figure 5). This operator samples from each distribution, and takes the lub of the resulting maps.

Sequencing (C_1 *to* $x.C_2$) is similar to application. We sample a map μ from C_1. As with application, there is only a finite number of distinct values v in $\mathit{dom}(\mu)$. For each unique v, we extend ρ, and evaluate $[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v)$, producing a finite set of distributions, which we flatten with Ξ.

The use of Ξ is well-defined by the following lemma:

Lemma 5.1 (Lubs of finite sets). *Let* $\{m_1, \ldots, m_n\} \subseteq [\![T]\!]$ *for some type* T*, then* $\Xi\{m_1, \ldots, m_n\}$ *is the least upper bound of* $\{m_1, \ldots, m_n\}$*.*

The proof is a straightforward induction on the size of the set. Because Ξ computes the least upper bound, it is independent of the order of $\{m_1, \ldots, m_n\}$. However, it assumes that the input set is finite, i.e., that there are only finitely many unique functions f. This assumption is discharged by Theorem 5.2 (see Section 5.3).

A further theorem (Theorem 5.5) makes the semantics well-defined. In particular, it ensures that the least upper bound in the definition of iteration exists and that the types ascribed to the denotations in Figure 5 are valid. The next sections discuss each theorem in turn. The proofs of these theorems can be found in Appendix ??.

5.3 Finite Maps

The function Ξ is only defined for *finite* sets. Therefore, we must verify that this function is only ever applied to a finite set. Informally, this is the case if we ensure that in Figure 5, μ is only bound to maps that have a finite domain. More formally, we desire the following:

Theorem 5.2 (Finite Maps). *For all computations* C*, such that* $\Gamma \vdash_c C : T$*,* $\rho \in [\![\Gamma]\!]$ *and* $A \in 2^{PH}$*, we have that for any* ω*QBS morphism* $f : X \to \mathbb{R}$*, where* $[\![T]\!] = 2^{PH} \to D(X)$*:*

$$\int_X f\, d([\![C]\!]\ \rho\ A) = \int_X \chi_F\, f\, d([\![C]\!]\ \rho\ A)$$

where $F = \{g \in X \mid dom(g) \text{ is finite}\}$, and $\chi_F : X \to \{0, 1\}$ is its characteristic function.

To prove this theorem we rely on logical relations [Tait 1967] $\mathcal{F}_S$ to define a stronger theorem (Theorem 5.4). The actual logical relations are $\mathcal{F}_S$ and $\mathcal{F}_T$, defined as follows:

Definition 5.3. The predicates $\mathcal{F}_S$ and $\mathcal{F}_T$ where S and T are value, respectively computations types, are defined inductively as:

$$\mathcal{F}_S = \begin{cases} [\![S]\!] & \text{if } S \neq \mathcal{T}\,T \\ \mathcal{F}_T & \text{if } S = \mathcal{T}\,T \end{cases}$$

$$\mathcal{F}_T = \left\{ f \in [\![T]\!] \;\middle|\; \forall g \in G(T), \forall A \in 2^{PH} : \int g \; d(f(A)) = \int \chi_{F(T)} \; g \; d(f(A)) \right\}$$

where

$$\begin{aligned} F(\mathcal{P}\,S) &= \{\mu : \mathcal{F}_S \rightharpoonup 2^{PH} \mid dom(\mu) \text{ is finite}\} \\ F(S \to T) &= \{\mu : (\mathcal{F}_S \to \mathcal{F}_T) \rightharpoonup 2^{PH} \mid dom(\mu) \text{ is finite}\} \\ G(\mathcal{P}\,S) &= ([\![S]\!] \rightharpoonup 2^{PH}) \to \mathbb{R} \\ G(S \to T) &= (([\![S]\!] \to [\![T]\!]) \rightharpoonup 2^{PH}) \to \mathbb{R} \end{aligned}$$

The idea of these logical relations is to restrict the denotations to those semantic objects where all partial maps have a finite domain. For instance, the denotation of non-thunk values never contains a map, hence $\mathcal{F}_S$ is simply $[\![S]\!]$ in this case. Thunked computations can contain computations, thus we restrict them to $\mathcal{F}_T$.

In a somewhat roundabout fashion, $\mathcal{F}_T$ says that the denotations of a computation of type T must only contain finite maps. In particular, it says that integrating any g with respect to $f(A)$ should be the same as integrating $\chi_{F(T)}\,g$ with respect to $f(A)$, where $F(T)$ only contains the finite maps of the appropriate type.[11] In other words, integrating while filtering out the infinite maps should not make a difference at all, meaning that the only maps that are present are finite. The following theorem, of which Theorem 5.2 is a corollary, states that every well-typed term's denotation is a member of the appropriate relation:

THEOREM 5.4. *For all values V and computations C,*

$$\textit{if } \Gamma \vdash_v V : S \textit{ and } \Gamma \vdash_c C : T, \textit{ then } [\![V]\!]\,\rho \in \mathcal{F}_S \textit{ and } [\![C]\!]\,\rho \in \mathcal{F}_T$$

where $\rho \in [\![\Gamma]\!]$ such that if $x : S_x \in \Gamma$, then $\rho(x) \in \mathcal{F}_{S_x}$.

PROOF. (sketch) By induction on the structure of the typing derivation, performing case analysis on the final rule application. The proof uses an equational reasoning style. □

5.4 Continuity

Continuity is a property that intuitively means that a function preserves the least upper bounds of the domain it operates on. In other words, it preserves the ωCPO structure. This is a technical requirement to ensure that the least-upper bound of the semantics of iteration exists. More formally, we desire the following property:

[11] The juxtaposition $\chi_{F(T)}\,g$ means pointwise multiplication.

Theorem 5.5 (Continuity). *For all computations C, such that $\Gamma \vdash_c C : T$, for all $\rho \in [\![\Gamma]\!]$, $[\![C]\!]\ \rho\ A$ is continuous in $A \in 2^{PH}$, i.e. $[\![C]\!]\ \rho\ A$ is monotone, and for all $A_1 \subseteq A_2 \subseteq \ldots \subseteq PH$:*

$$\bigsqcup_{i \geq 1}([\![C]\!]\ \rho\ A_i) = [\![C]\!]\ \rho\ \big(\bigcup_{i\geq 1} A_i\big)$$

To prove this theorem, we define the following logical relation:

Definition 5.6. The predicates $\mathcal{C}_S$ and $\mathcal{C}_T$ where S and T are value, respectively computations types, are defined inductively as:

$$\mathcal{C}_S = \begin{cases} [\![S]\!] & \text{if } S \neq \mathcal{T}\,T \\ \mathcal{C}_T & \text{if } S = \mathcal{T}\,T \end{cases}$$

$$\mathcal{C}_T = \left\{ f \in [\![T]\!] \;\middle|\; \begin{array}{l} f \text{ is continuous, and} \\ \forall g \in G(T), \forall A \in 2^{PH} : \int g\ d(f(A)) = \int \chi_{C(T)}\ g\ d(f(A)) \end{array} \right\}$$

where

$$\begin{aligned} C(\mathcal{P}\,S) &= \{\mu : \mathcal{C}_S \rightharpoonup 2^{PH}\} \\ C(S \to T) &= \{\mu : (\mathcal{C}_S \to \mathcal{C}_T) \rightharpoonup 2^{PH}\} \\ G(\mathcal{P}\,S) &= ([\![S]\!] \rightharpoonup 2^{PH}) \to \mathbb{R} \\ G(S \to T) &= (([\![S]\!] \to [\![T]\!]) \rightharpoonup 2^{PH}) \to \mathbb{R} \end{aligned}$$

Similar to Section 5.3 the idea is to restrict the denotations to those semantics objects that are continuous (in the input set—not the environment), and for those partial maps that have a function domain, the domain only contains continuous functions. Note that we do not require that the partial maps *themselves* are continuous. Indeed, this is clearly not the case (e.g. in the case of *produce* V and $\lambda x\!:\!S.C$). The denotations for non-thunk values do not contain input sets or maps, hence $\mathcal{C}_S$ is simply $[\![S]\!]$. Thunked computations contain computations, thus $\mathcal{C}_{\mathcal{T}\,T} = \mathcal{C}_T$. For computations of type T, $\mathcal{C}_T$ says that the semantics itself must be continuous, and any element of a partial map's domain must be continuous.

We prove the following theorem, of which Theorem 5.5 is a direct consequence:

Theorem 5.7. *For all values V and computations C:*

$$\textit{if } \Gamma \vdash_v V\!:\!S \textit{ and } \Gamma \vdash_c C\!:\!T, \textit{ then } [\![V]\!]\ \rho \in \mathcal{C}_S \textit{ and } [\![C]\!]\ \rho \in \mathcal{C}_T$$

where $\rho \in [\![\Gamma]\!]$ such that if $x : S_x \in \Gamma$, then $\rho(x) \in \mathcal{C}_{S_x}$.

5.5 Conservativity

The next theorem relates the semantics to the original PNK semantics, showing that our semantics behaves identical to the original PNK semantics on probabilistic computations.

Theorem 5.8 (Conservativity). *Let C be a closed probabilistic computation, and let $[\![C]\!]_{PNK}$ be the denotation of C in the probabilistic PNK semantics [Smolka et al. 2017b], re-translated into ωQBS (see Appendix **??**), then for all $A \in 2^{PH}$:*

$$[\![C]\!]_{PNK}\ A = [\![C]\!]\ ()\ A \ggg\!= \lambda\mu \to \textit{return}\ (\mu(()))$$

Summary. We have a *well-defined* denotational semantics for PλωNK. This semantics is a *conservative* extension of PNK's semantics. However, it is not clear how to compute this semantics. Recall that application $C\ V$ applies every *unique* $f \in dom(\mu)$ *exactly* once, in parallel. Because parallel composition is not idempotent [Foster et al. 2016], duplicate applications are not innocent. The situation for sequencing is analogous. To the best of our knowledge, there is no decidable procedure for this uniqueness problem. We could follow Smolka et al. [2017a, 2019] and remove *dup*, making the state-space finite and discrete. Although a perfectly valid solution, we believe that this restricts the properties we can model in our language too much (e.g., modelling latency is not possible). Instead, we restrict the type of parallel composition to $\mathcal{P}\,\mathbf{1}$, to ensure that $dom(\mu)$ contains a single element. This enables the compilation of closed PλωNK terms into PNK. The program is then approximated with PNK's approximation procedure. We discuss this approach in detail in Section 6.

6 COMPILATION TO PNK

Let us make precise the restriction referred to in the previous section. We replace the judgements $\Gamma \vdash_v V : S$ and $\Gamma \vdash_c C : T$ with $\Gamma \Vdash_v V : S$ and $\Gamma \Vdash_c C : T$. The rules for these judgements are analogous to the original rules, except that we replace

$$\frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 : T} \quad \text{with} \quad \frac{\Gamma \Vdash_c C_1 : \mathcal{P}\,\mathbf{1} \qquad \Gamma \Vdash_c C_2 : \mathcal{P}\,\mathbf{1}}{\Gamma \Vdash_c C_1 \,\&\, C_2 : \mathcal{P}\,\mathbf{1}}$$

The new rule restricts the type of parallel composition to $\mathcal{P}\,\mathbf{1}$. In other words, the value of & is completely predictable: it must be *unit*. Furthermore, parallelism is the only way to grow the domain of the finite maps. Thus, all domains are now either a single function, or a single value. From a different perspective, we have just restricted the parallelism of PλωNK to the parallelism present in PNK.

6.1 Elaboration

We only compile closed computations of type $\mathcal{P}\,\mathbf{1}$. This is reasonable because complete PλωNK models are not functions and have no free variables. Initially, we only deal with *terminal* computations (Section 3.1). By case analysis on the typing judgement, such computations are either *produce unit* or probabilistic computations (i.e. only consist of PNK terms). The following relation elaborates these computations into PNK:

Definition 6.1 (Elaboration). Define $R \rightsquigarrow R$ as:

$$\begin{array}{rcll}
P & \rightsquigarrow & P & \\
V \leftarrow V & \rightsquigarrow & V \leftarrow V & \\
dup & \rightsquigarrow & dup & \\
R_1 \,;\, R_2 & \rightsquigarrow & E_1 \,;\, E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\
R_1 \,\&\, R_2 & \rightsquigarrow & E_1 \,\&\, E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\
R_1 \oplus R_2 & \rightsquigarrow & E_1 \oplus E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\
R^* & \rightsquigarrow & E^* & \text{if } R \rightsquigarrow E \\
produce\ V & \rightsquigarrow & skip & \\
\lambda x : S.C & \rightsquigarrow & skip &
\end{array}$$

Elaboration preserves the semantics of a closed term of type $\mathcal{P}\,\mathbf{1}$:

THEOREM 6.2 (SOUNDNESS OF ELABORATION). *Let R_1, R_2 be terminals such that $\Vdash_c R_1 : \mathcal{P}\,\mathbf{1}$, $\Vdash_c R_2 : \mathcal{P}\,\mathbf{1}$ and $R_1 \rightsquigarrow R_2$, then $[\![R_1]\!] = [\![R_2]\!]$.*

Moreover, the elaboration always exists for closed terms of the right type:

$$\boxed{C \Downarrow R}$$

$$\frac{}{P \Downarrow P} \qquad \frac{}{F \leftarrow N \Downarrow F \leftarrow N} \qquad \frac{}{\mathit{dup} \Downarrow \mathit{dup}} \qquad \frac{}{\mathit{produce}\ V \Downarrow \mathit{produce}\ V}$$

$$\frac{}{\lambda x\!:\!S.C \Downarrow \lambda x\!:\!S.C} \qquad \frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1\,;C_2 \Downarrow R_1\,;R_2} \qquad \frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1 \,\&\, C_2 \Downarrow R_1 \,\&\, R_2}$$

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1 \oplus C_2 \Downarrow R_1 \oplus R_2} \qquad \frac{C \Downarrow R}{C^* \Downarrow R^*} \qquad \frac{C \Downarrow R}{\mathit{force}\ (\mathit{thunk}\ C) \Downarrow R}$$

$$\frac{C_1 \Downarrow \lambda x\!:\!S.C_{11} \qquad [x \mapsto V]C_{11} \Downarrow R}{C_1 V \Downarrow R} \qquad \frac{C_1 \Downarrow R_{11}\,;R_{12} \qquad R_{12}\ V \Downarrow R_2}{C_1\ V \Downarrow R_{11}\,;R_2}$$

$$\frac{C_1 \Downarrow \mathit{produce}\ V \qquad [x \mapsto V]C_2 \Downarrow R}{C_1\ \mathit{to}\ x.C_2 \Downarrow R} \qquad \frac{C_1 \Downarrow R_{11} \oplus R_{12} \qquad R_{11}\ V \Downarrow R_1 \qquad R_{12}\ V \Downarrow R_2}{C_1\ V \Downarrow R_1 \oplus R_2}$$

$$\frac{C_1 \Downarrow P \qquad [x \mapsto \mathit{unit}]C_2 \Downarrow R}{C_1\ \mathit{to}\ x.C_2 \Downarrow P\,;R} \qquad \frac{C_1 \Downarrow F \leftarrow N \qquad [x \mapsto \mathit{unit}]C_2 \Downarrow R}{C_1\ \mathit{to}\ x.C_2 \Downarrow F \leftarrow N\,;R}$$

$$\frac{C_1 \Downarrow \mathit{dup} \qquad [x \mapsto \mathit{unit}]C_2 \Downarrow R}{C_1\ \mathit{to}\ x.C_2 \Downarrow \mathit{dup}\,;R} \qquad \frac{C_1 \Downarrow {R_{11}}^* \qquad [x \mapsto \mathit{unit}]C_2 \Downarrow R_2}{C_1\ \mathit{to}\ x.C_2 \Downarrow {R_{11}}^*\,;R_2}$$

$$\frac{C_1 \Downarrow R_{11}\,;R_{12} \qquad R_{12}\ \mathit{to}\ x.C_2 \Downarrow R_2}{C_1\ \mathit{to}\ x.C_2 \Downarrow R_{11}\,;R_2} \qquad \frac{C_1 \Downarrow R_{11} \,\&\, R_{12} \qquad [x \mapsto \mathit{unit}]C_2 \Downarrow R_2}{C_1\ \mathit{to}\ x.C_2 \Downarrow (R_{21} \,\&\, R_{22})\,;R_2}$$

$$\frac{C_1 \Downarrow R_{11} \oplus R_{12} \qquad R_{11}\ \mathit{to}\ x.C_2 \Downarrow R_{21} \qquad R_{12}\ \mathit{to}\ x.C_2 \Downarrow R_{22}}{C_1\ \mathit{to}\ x.C_2 \Downarrow R_{21} \oplus R_{22}}$$

Fig. 6. Rules for reduction from P$\lambda\omega$NK to PNK.

✓Theorem 6.3 (Completeness of Elaboration). *Let R be a terminal, then there exists precisely one probabilistic E such that $R \rightsquigarrow E$.*

6.2 Reduction

Converting a closed terminal into a PNK program is only half the battle. The other half is performed by the bigstep relation $C \Downarrow R$ given in Figure 6. It reduces a computation to a terminal. We can make the following observations about the bigstep relation:

- Atomic computations simply reduce to themselves.
- Compound computations built with PNK operations (sequential and parallel composition, probabilistic choice and iteration) are reduced to the reduction of their subcomputations.
- Forcing a thunk reduces to the reduct of the thunked computation.
- Application of a lambda substitues the value into the body of the lambda. When the first argument reduces to sequential composition or probabilistic choice, the application

distributes over this argument. This is *not* possible for parallel composition, since it is not distributive [Foster et al. 2016].

- When C_1 reduces to *produce* V, sequencing (C_1 *to* $x.C_2$) substitutes V for x in C_2, reducing the result of the substitution. When the first argument instead reduces to a predicate, a modification, a duplication, parallel composition or an iteration, C_1 always reduces to a terminal of type $\mathcal{P}\,\mathbf{1}$. Hence, we substitute *unit* for V in these cases. When the first argument reduces to sequential composition composition or probabilistic choice, sequencing, like application also *distributes* over this argument. Finally, if the first argument reduces to parallel composition, we know that the value it produces *must* be unit, and so we can always substitute *unit* for x.

To be clear, our compilation strategy is as follows: (1) *Reduce* a PλωNK computation to a terminal, and (2) *elaborate* the remaining terminal into PNK.

To ensure that our compilation delivers correct results, it remains to show that step (1) terminates, and that this step is sound. Soundness means that the denotational semantics of the program is preserved. It is expressed by the following theorem:

THEOREM 6.4 (SOUNDNESS OF REDUCTION). *Let C, R be computations, if $\Gamma \Vdash_c C:T$ and $C \Downarrow R$ then $[\![C]\!] = [\![R]\!]$.*

Termination is our subsequent concern. It follows from strong normalisation of the reduction relation, a property of the meta-theory of reduction, discussed in the next section.

6.3 Meta-theory of Reduction

The reduction relation obeys the standard type-preservation theorems, and in addition, is *strongly normalising*. In detail, terminals are reduced to themselves (Theorem 6.5). Moreover, all reductions result in terminals (by definition), in a deterministic fashion (Theorem 6.6) and preserve types (Theorem 6.7). Finally, reduction is strongly normalising (Theorem 6.8).

✓THEOREM 6.5 (REFLECTION). *Let R be a terminal, then $R \Downarrow R$.*

✓THEOREM 6.6 (DETERMINACY). *Let C, R_1, R_2 be computations, if $C \Downarrow R_1$ and $C \Downarrow R_2$, then $R_1 = R_2$.*

✓THEOREM 6.7 (PRESERVATION). *Let C, R be computations, if $\Gamma \Vdash_c C:T$ and $C \Downarrow R$, then $\Gamma \vdash_c R:T$. Conversely, if $\Gamma \Vdash_c C:T_1$, $\Gamma \Vdash_c R:T_2$ and $C \Downarrow R$, then $T_1 = T_2$.*

✓THEOREM 6.8 (STRONG NORMALISATION). *Let C be a computation such that $\Vdash_c C:T$ for some computation type T, then there exists a terminal R such that $C \Downarrow R$.*

In addition to the theorems shown here, PλωNK also satisfies additional inversion and substitution lemmas that are instrumental in proving these theorems.

The meta-theory discussed in this section has been mechanised with the aid of the Abella proof-assistant [Gacek 2008]. The proofs for these theorems proceed by induction either on the structure of terminals or the structure of the bigstep relation. Each of proof has many cases that need to be checked. By using a theorem prover, we ensure that no cases or conditions are forgotten.

The most involved proof is Strong Normalisation, which requires a logical-relation style proof technique [Tait 1967]. This particular proof was inspired by the standard proof of strong normalisation of CBPV, described in Levy's thesis [2001].

The definition of the logical relations are shown in Figure 7. In essence, we need to define three mutually-recursive type-indexed logical relations: one for values ($\mathbf{V}[S]$), one for closed

$$\mathbf{V}[\mathbf{1}] = \{\mathit{unit}\} \quad \mathbf{V}[\mathcal{H}] = \mathit{Headers} \quad \mathbf{V}[\mathbb{N}] = \mathbb{N} \quad \mathbf{V}[\mathcal{T}\,T] = \{\mathit{thunk}\ C \mid C \in \mathbf{C}[T]\}$$

$$\begin{array}{rl}
R \in \mathbf{T}[\mathcal{P}\,\mathbf{1}] & \text{iff } \Vdash_c R{:}\mathcal{P}\,\mathbf{1} \text{ where } R \text{ is atomic} \\
R^* \in \mathbf{T}[\mathcal{P}\,\mathbf{1}] & \text{iff } R \in \mathbf{T}[\mathcal{P}\,\mathbf{1}] \text{ and } \Vdash_c R^*{:}\mathcal{P}\,\mathbf{1} \\
\mathit{produce}\ V \in \mathbf{T}[\mathcal{P}\,S] & \text{iff } V \in \mathbf{V}[S] \text{ and } \Vdash_c \mathit{produce}\ V{:}\mathcal{P}\,S \\
R_1\,;R_2 \in \mathbf{T}[\mathcal{P}\,S] & \text{iff } \exists T' : R_1 \in \mathbf{T}[T'] \text{ and } R_2 \in \mathbf{T}[\mathcal{P}\,S] \text{ and } \Vdash_c R_1\,;R_2{:}\mathcal{P}\,S \\
R_1 \,\&\, R_2 \in \mathbf{T}[\mathcal{P}\,S] & \text{iff } R_1, R_2 \in \mathbf{T}[\mathcal{P}\,S] \text{ and } \Vdash_c R_1 \,\&\, R_2{:}\mathcal{P}\,S \\
R_1 \oplus R_2 \in \mathbf{T}[\mathcal{P}\,S] & \text{iff } R_1, R_2 \in \mathbf{T}[\mathcal{P}\,S] \text{ and } \Vdash_c R_1 \oplus R_2{:}\mathcal{P}\,S \\
R \in \mathbf{T}[S \to T] & \text{iff } R \text{ is terminal and } \Vdash_c R{:}S \to T \text{ and } \forall V \in \mathbf{V}[S] : (R\ V) \in \mathbf{C}[T]
\end{array}$$

$$C \in \mathbf{C}[T] \text{ iff } \Vdash_c C{:}T \text{ and } \exists R \in \mathbf{T}[T] : C \Downarrow R$$

Fig. 7. Logical relations involved in proving Strong Normalisation

terminal computations ($\mathbf{T}[T]$) and one for all computations, terminal or non-terminal ($\mathbf{C}[T]$). The idea is that those relations contain only closed computations for which the bigstep relation $\Downarrow$ terminates. For values, it contains all closed non-thunk values, and only closed thunks of terminating computations. Additionally, computations of function type must preserve termination when applied to terminating values.

Compared to Levy's logical relations, our logical relation does not contain product or sum types. In our proof we leverage standard Abella techniques to prove theorems with logical relations, which requires defining and proving substitution lemmas for every syntactic form. Moreover, we also need to show additional preservation lemmas for sequential and parallel composition, and for probabilistic choice.

The proof of strong normalisation, together with the definitions of the logical relations and supporting lemmas amounts to a little under 800 lines of Abella code (roughly 1500 LOC in total).

6.4 Discussion

We have identified a class of P$\lambda\omega$NK programs that can be safely compiled to PNK. In particular, the class consists of computations C such that $\Vdash_c C{:}\mathcal{P}\,\mathbf{1}$, i.e., those programs that do not use parallelism beyond what is present in PNK. It should be possible to ease this restriction slightly. For instance, unlike functions, we can distinguish between distinct headers and literals. This leads to only finitely many cases, which could be encoded explicitly into PNK.

Once a program has been compiled, it can be approximated in PNK. The approximation proceeds by expanding iterations in the program up to n times, for finite n. More iterations improve the accuracy of the results [Smolka et al. 2017b].

7 RELATED WORK

Software Defined Networks. Software Defined Networks (SDN) [Foster et al. 2013] aim to decrease the complexity of the modern computer networking environment, by offering a clean open interface between heterogeneous networking devices (e.g. routers, switches and firewalls). This is accomplished through the OpenFlow[12] protocol. Unfortunately, this is a rather low-level protocol making it inconvenient to program hardware in OpenFlow directly.

The aim of the Frenetic project [Reich et al. 2013] is to design the right high-level abstractions for controlling OpenFlow hardware. NetKAT [Anderson et al. 2014] and PNK [Foster et al. 2016] were developed under this project. Some preliminary case studies were performed with PNK, modelling the behaviour of several traffic engineering approaches. The effort involved in these case studies was not reported. Interestingly, the few samples of code that were provided seem to use features (e.g. finite iterations, variables), that are not part of the formalised fragment of PNK, but could be implemented fully within PλωNK.

Probabilistic Programming. The goal of probabilistic programming [Goodman 2013] can be captured by the following equation:

$$\text{PPL} = \text{MODELLING LANGUAGE} + \text{INFERENCE ALGORITHM}$$

That is, probabilistic programming's goal is to unify probabilistic modelling and general purpose programming: probabilistic models are written in the language, and the probabilities are inferred using a generic inference algorithm.

Stan [Carpenter et al. 2017] is a very popular statistical modelling language, with bindings to R, Python, MATLAB, Julia and several others. More recently, languages such as Gen [Cusumano-Towner et al. 2019] and Turing [Ge et al. 2018] have started to make the inference algorithms programmable, in addition to the model, since fine-tuning the inference can lead to large performance gains.

Probabilistic Programming has been applied to such diverse problems as 3D body pose estimation from depth data [Cusumano-Towner et al. 2019], genetics [De Maeyer et al. 2013] and Automatic Video Montage [Aerts et al. 2016].

Functional PPLs such as Anglican [Wood et al. 2014], Venture [Lu 2016], or Gen [Lu 2016], allow distributions over higher-order functions. However, unlike this work, they do not formalise the semantics of higher-order functions, focusing instead on language design and implementation, and inference. The work on quasi-Borel Spaces is at least partially motivated by the unfilled need for a theoretical foundation for these languages [Heunen et al. 2017]. Quasi-Borel Spaces have been used by Scibior et al. [2018] to verify the correctness of modular Bayesian inference algorithms.

Probabilistic powerdomains have been extensively investigated (see e.g., Bacci et al. [2018]; Battenfeld et al. [2007]; Goubault-Larrecq and Varacca [2011]; Jones and Plotkin [1989]; Jung and Tix [1998]; Saheb-Djahromi [1980]). Nevertheless, until the work of Vákár et al. [2019], a convenient continuous probabilistic powerdomain that supports iteration and is commutative proved elusive. However, the ω-quasi Borel Spaces are not the only approach to this problem, as we remark in the the next paragraph.

Probabilistic Call-By-Push-Value. Although PλωNK does not model CBPV exactly, it was heavily inspired by it. CBPV was developed by Levy [2001] as a paradigm that subsumes both CBV and CBN. Since then, Ehrhard and Tasson [2019] have developed a probabilistic CBPV calculus. A technical difference is that they give a semantics in terms of probabilistic coherence spaces [Danos and Ehrhard 2011], whereas we use a monadic semantics based on ωQBSes.

Goubault-Larrecq [2019] cleverly side-steps the issue of providing a commutative statistical higher-order powerdomain, by giving a semantics for CBPV that interprets value types and

[12] https://www.opennetworking.org/

computation types differently. The values are interpreted in a category which is closed under the powerdomain functor, and the computations are interpreted as DCPOs. His language also has demonic non-determinism, statistical termination testers and parallel if statements.

A crucial difference with our work is that monadic state is not present in either calculus, while it is in ours. This significantly complicates our denotational semantics, in particular for application and sequencing. Moreover, these calculi do observe the isomorphism mentioned in Section 5.1. A detailed investigation into the relationships between these calculi and our language is reserved for future work.

8 CONCLUSIONS AND FUTURE WORK

In future work, we intend to quantitatively evaluate the impact P$\lambda\omega$NK, by re-implementing the existing PNK case-studies, and study the improvements in terms of readability and maintainability.

To support this quantitative study, we need to further develop and optimise our prototype. We believe that performance can be improved by incorporating techniques such as knowledge compilation [Kisa et al. 2014; Smolka et al. 2015] and defunctionalisation [Danvy and Nielsen 2001; Reynolds 1998].

Furthermore, our work contains a significant amount of paper proofs. These proofs are inductive proofs, using equational reasoning and logical relations, which should not be too difficult to mechanise. However, the requisite background theory, i.e., (ω-)Quasi-Borel Spaces, has to be mechanised first.

Lastly, we are investigating reformulations of P$\lambda\omega$NK for other paradigms such as true CBPV and Fine-Grained CBV [Levy 2001, App. A.3]. The key difference appears to be that function typed terms should only evaluate their side-effects when applied to a value.

Conclusion. In this article we presented P$\lambda\omega$NK, a functional network modelling language that combines state, parallelism and probabilistic choice. Because it combines higher-order functions and probability, we cannot give it a purely measure-theoretic semantics. Instead, we leverage ω-Quasi Borel Spaces to define our denotational semantics. We also define a strongly normalising type system for P$\lambda\omega$NK. Since the main purpose of P$\lambda\omega$NK is verification, the additional flexibility of general recursion is not required. Indeed, strong normalisation is necessary to make our semantics well-defined. Moreover, we develop a procedure to compile programs in our language to the simpler language Probabilistic NetKAT, given small type restrictions.

ACKNOWLEDGMENTS

We would like to thank Ohad Kammar and Mathijs Vákár for personally explaining ωQBSes and providing us with an early draft of their paper [Vákár et al. 2019].

We are grateful to the anonymous POPL reviewers for their constructive feedback and helpful in-depth comments.

Alexander Vandenbroucke is an SB Fellow of the flemish Fund for Scientific Research(FWO), File No.: 1S68117N. This work is further supported by FWO Grant No. G095917N.

REFERENCES

Bram Aerts, Toon Goedemé, and Joost Vennekens. 2016. A Probabilistic Logic Programming Approach to Automatic Video Montage. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence (ECAI'16)*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 234–242. https://doi.org/10.3233/978-1-61499-672-9-234

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetkAT: semantic foundations for networks. In *POPL*. ACM, 113–126.

Robert J Aumann et al. 1961. Borel structures for function spaces. *Illinois Journal of Mathematics* 5, 4 (1961), 614–630.

Giorgio Bacci, Robert Furber, Dexter Kozen, Radu Mardare, Prakash Panangaden, and Dana Scott. 2018. Boolean-Valued Semantics for the Stochastic λ-Calculus. In *LICS*. ACM, 669–678.

Ingo Battenfeld, Matthias Schröder, and Alex Simpson. 2007. A Convenient Category of Domains. *Electr. Notes Theor. Comput. Sci.* 172 (2007), 69–99.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI*. ACM, 221–236.

Vincent Danos and Thomas Ehrhard. 2011. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Inf. Comput.* 209, 6 (2011), 966–991.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *PPDP*. ACM, 162–174.

Dries De Maeyer, Joris Renkens, Lore Cloots, Luc De Raedt, and Kathleen Marchal. 2013. PheNetic: network-based interpretation of unstructured gene lists in E. coli. *Molecular BioSystems* 9, 7 (2013), 1594–1603.

Thomas Ehrhard and Christine Tasson. 2019. Probabilistic call by push value. *Logical Methods in Computer Science* 15, 1 (2019).

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP* 15, 3 (2015), 358–401.

Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, et al. 2013. Languages for software-defined networks. *IEEE Communications Magazine* 51, 2 (2013), 128–134.

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *IJCAR (Lecture Notes in Computer Science)*, Vol. 5195. Springer, 154–161.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*. 1682–1690.

Noah D. Goodman. 2013. The principles and practice of probabilistic programming. In *POPL*. ACM, 399–402.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2012. Church: a language for generative models. *CoRR* abs/1206.3255 (2012).

Jean Goubault-Larrecq. 2019. A Probabilistic and Non-Deterministic Call-by-Push-Value Language. In *Thirty-Fourth Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, Canada, June 24–27, 2019, Proceedings*. To appear. https://arxiv.org/pdf/1812.11573.pdf

Jean Goubault-Larrecq and Daniele Varacca. 2011. Continuous Random Variables. In *LICS*. IEEE Computer Society, 97–106.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *LICS*. IEEE Computer Society, 1–12.

Claire Jones and Gordon D Plotkin. 1989. A probabilistic powerdomain of evaluations. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. IEEE, 186–195.

Achim Jung and Regina Tix. 1998. The troublesome probabilistic powerdomain. *Electr. Notes Theor. Comput. Sci.* 13 (1998), 70–91.

Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. 2014. Probabilistic Sentential Decision Diagrams. In *KR*. AAAI Press.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.

Paul Blain Levy. 2001. *Call-by-push-value.* Ph.D. Dissertation. Queen Mary University of London, UK.

Anthony Lu. 2016. *Venture: An extensible platform for probabilistic meta-programming.* Master's thesis. Massachusetts Institute of Technology.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular SDN programming with Pyretic. *Technical Report of USENIX* (2013).

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397.

Nasser Saheb-Djahromi. 1980. CPO's of measures for nondeterminism. *Theoretical Computer Science* 12, 1 (1980), 19–37.

Adam Scibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *PACMPL* 2, ICFP (2018), 83:1–83:29.

Steffen Smolka, Spiridon Aristides Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. In *ICFP*. ACM, 328–341.

Steffen Smolka, David M. Kahn, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017a. Deciding Probabilistic Program Equivalence in NetKAT. *CoRR* abs/1707.02772 (2017). arXiv:1707.02772 http://arxiv.org/abs/1707.02772

Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017b. Cantor meets Scott: semantic foundations for probabilistic networks. In *POPL*. ACM, 557–571.

Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable verification of probabilistic networks. In *PLDI*. ACM, 190–203.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 855–879.

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212.

Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *PACMPL* 3, POPL (2019), 36:1–36:29. https://dl.acm.org/citation.cfm?id=3290349

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *AISTATS (JMLR Workshop and Conference Proceedings)*, Vol. 33. JMLR.org, 1024–1032.

A ATOMIC COMPUTATIONS

The denotational semantics predicates, modifications and duplications all share a very similar structure. Essentially, they immediately return a partial map from the unit value to a set, that is somehow a transformation of the input set. We call these *atomic* computations (see Smolka et al. [2017b]).

This property is very helpful, since, for the purpose of proving properties about their denotational semantics, atomic computations can be treated uniformly.

To make this more precise, we first define how individual elements of the input set are transformed. For predicates, there is a boolean function B_P that decides if an element of the input is retained in the output:

Definition A.1. Let P be a predicate such that $\Gamma \vdash_c P : \mathcal{P}\,\mathbf{1}$, then $B_P : [\![\Gamma]\!] \to PH \to \{0, 1\}$ is defined as:

$$
\begin{aligned}
B_{skip}\ \rho\ x &= 1 \\
B_{drop}\ \rho\ x &= 0 \\
B_{V_1 = V_2}\ \rho\ (\pi :: h) &= \pi.[\![V_1]\!]\ \rho = [\![V_2]\!]\ \rho \\
B_{P_1 \wedge P_2}\ \rho\ x &= B_{P_1}\ \rho\ x \wedge B_{P_2}\ \rho\ x \\
B_{P_1 \vee P_2}\ \rho\ x &= B_{P_1}\ \rho\ x \vee B_{P_2}\ \rho\ x \\
B_{\neg P}\ \rho\ x &= \neg(B_P\ \rho\ x)
\end{aligned}
$$

Lemma A.2. *Let P e a predicate such that $\Gamma \vdash_c P : \mathcal{P}\,\mathbf{1}$, $\rho \in [\![\Gamma]\!]$ and $A \in 2^{PH}$: then*

$$[\![p]\!]^p\ \rho\ A = \{x \in A \mid B_P\ \rho\ x\}$$

Proof. By straightforward induction on the structure of the typing derivation. □

For atomic computations there is a function f_P that transforms the input elements:

Definition A.3. Let C be an atomic computation, such that $\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}$ then $f_C : [\![\Gamma]\!] \to PH \rightharpoonup PH$ is defined as:

$$
\begin{aligned}
f_P\ \rho\ x &= \begin{cases} x & \text{if } B_P\ \rho\ x \\ \perp & \text{otherwise} \end{cases} \\
f_{V_1 \leftarrow V_2}\ \rho\ (\pi :: h) &= \pi[[\![V_1]\!]\ \rho \mapsto [\![V_2]\!]\ \rho] :: h \\
f_{dup}\ \rho\ (\pi :: h) &= \pi :: \pi :: h
\end{aligned}
$$

Lemma A.4. *let C be an atomic computation such that $\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}$, then $[\![C]\!]\ \rho\ A = return\ (\lambda() \to \{f_C\ \rho\ x \mid x \in A\})$ for all $\rho \in [\![\Gamma]\!]$ and $A \in 2^{PH}$.*

Here it should be understood that if $f_C\ \rho$ is undefined on some element x, the element is not included in the resulting set.

Proof. By case analysis on the structure of the atomic computation C. If C is a predicate, the required follows from the definition of $[\![C]\!]\ \rho\ A$ and Lemma **??**. If C is a modification or duplication, the required follows immediately from the definition. □

B RE-TRANSLATION OF PNK SEMANTICS

Smolka et al. [2017b] give denotational semantics for deterministic and probabilistic PNK programs at the same time, by interpreting the same semantics in a different monad (the identity monad and the monad of (sub)probability measures, respectively). We only concern ourselves with the probabilistic semantics.

First note that they do not distinguish between $\wedge$ and ; or $\vee$ and &, which we do. For this reason we define $[\![\cdot]\!]_{PNK}$ as:

$$
\begin{aligned}
[\![C]\!]_{PNK} &: 2^{PH} \to D(2^{PH}) \\
[\![\mathit{drop}]\!]_{PNK}\ A &= \mathit{return}\ \emptyset \\
[\![\mathit{skip}]\!]_{PNK}\ A &= \mathit{return}\ A \\
[\![f = n]\!]_{PNK}\ A &= \mathit{return}\ (\{\pi :: h \in A \mid \pi.f = n\}) \\
[\![\neg P]\!]_{PNK}\ A &= [\![P]\!]_{PNK}\ A \gg\!= \lambda B \to \mathit{return}\ (B - A) \\
[\![P_1 \wedge P_2]\!]_{PNK}\ A &= [\![P_1]\!]_{PNK}\ A \gg\!= [\![P_2]\!]_{PNK} \\
[\![P_1 \vee P_2]\!]_{PNK}\ A &= [\![P_1]\!]_{PNK}\ A \gg\!= \lambda B \to [\![P_2]\!]_{PNK}\ A \gg\!= \lambda C \to \mathit{return}\ (B \cup C) \\
[\![f \leftarrow n]\!]_{PNK}\ A &= \mathit{return}\ (\{\pi[f \mapsto n] :: h \in A \mid \pi :: h \in A\}) \\
[\![\mathit{dup}]\!]_{PNK}\ A &= \mathit{return}\ (\{\pi :: \pi :: h \in A \mid \pi :: h \in A\}) \\
[\![C_1 \,;C_2]\!]_{PNK}\ A &= [\![C_1]\!]_{PNK}\ A \gg\!= [\![C_2]\!]_{PNK} \\
[\![C_1 \,\&\, C_2]\!]_{PNK}\ A &= [\![C_1]\!]_{PNK}\ A \gg\!= \lambda B \to [\![C_2]\!]_{PNK}\ A \gg\!= \lambda C \to \mathit{return}\ (B \cup C) \\
[\![C_1 \oplus r C_2]\!]_{PNK}\ A &= r([\![P_1]\!]\ A\ ) + (1-r)([\![P_2]\!]\ A\ ) \\
[\![C^*]\!]_{PNK}\ A &= \bigsqcup_{n \geq 0} [\![C^n]\!]_{PNK}\ A
\end{aligned}
$$

C PROOFS

C.1 Lemma 5.1

PROOF. Let $M = \{m_1, \ldots, m_n\}$. By induction on n, we show that ΞM is the least upper bound of M.

Case BASE $\quad n = 0$

If $n = 0$, then $M = \emptyset$ and the statement is vacuously true.

Case INDUCTION $\quad n + 1$

Observe that:

$$\Xi M = \Xi\{m_1, \ldots, m_n, m_{n+1}\} = \Xi\{\Xi\{m_1, \ldots, m_n\}, m_{n+1}\}$$

By induction, we know that $\Xi\{m_1, \ldots, m_n\}$ is the least upper bound of $\{m_1, \ldots, m_n\}$. Hence, we must only show that for any $m_1, m_2 \in D(X)$, $m_1 \Xi m_2$ is the least upper bound of m_1 and m_2. Recall that D is a continuation monad, then we can proceed as follows (except where otherwise noted, the proof proceeds by continuity):

$$
\begin{aligned}
& m_1 \Xi m_2 \\
=& m_1 \gg\!= \lambda x \to m_2 \gg\!= \lambda y \to return x \sqcup y \\
=& \lambda k \to m_1\ (\lambda x \to m_2\ (\lambda y \to k\ (x \sqcup y))) \\
=& \lambda k \to m_1\ (\lambda x \to m_2\ (\lambda y \to k\ x \sqcup k\ y)) \\
=& \lambda k \to m_1\ (\lambda x \to m_2\ ((\lambda y \to k\ x) \bigvee (\lambda y \to k\ y))) \\
=& \lambda k \to m_1\ (\lambda x \to \underbrace{m_2\ (\lambda y \to k\ x)}_{k\ x} \sqcup \underbrace{m_2\ (\lambda y \to k\ y)}_{m_2\ k}) \\
& \qquad \textit{m_2 is a probability distribution, hence, } \int dm_2 = 1 \textit{ and η-reduction} \\
=& \lambda k \to m_1\ (\lambda x \to k\ x \sqcup m_2\ k) \\
=& \lambda k \to m_1\ ((\lambda x \to k\ x) \bigvee (\lambda x \to m_2\ k)) \\
=& \lambda k \to \underbrace{m_1\ (\lambda x \to k\ x)}_{m_1\ k} \sqcup \underbrace{m_1\ (\lambda x \to m_2\ k)}_{m_2\ k} \\
& \qquad \textit{η-reduction and m_1 is a probability distribution, hence, } \int dm_2 = 1 \\
=& \lambda k \to m_1\ k \sqcup m_2\ k \\
=& (\lambda k \to m_1\ k) \sqcup (\lambda k \to m_2\ k)
\end{aligned}
$$

□

C.2 Theorem 5.4

PROOF. By induction on the structure of the typing derivation, performing case analysis on the final rule application.

In what follows, let $\Gamma \vdash_v V : S$, $\Gamma \vdash_c C : T$, and $\rho \in [\![\Gamma]\!]$, such that $\forall (x : S_x) \in \Gamma : \rho(x) \in \mathcal{F}_{S_x}$.

Case T-VAR $\qquad V = x_i \qquad \dfrac{x_i : S_i \in \Gamma}{\Gamma \vdash_v x_i : S_i}$

By assumption, $[\![x_i]\!]\ \rho = \rho(x_i) \in \mathcal{F}_{S_i}$.

Case T-UNIT $\qquad V = unit \qquad \dfrac{}{\Gamma \vdash_v unit : \mathbf{1}}$

By assumption, $[\![unit]\!]\ \rho = \ \in \{()\} = [\![\mathbf{1}]\!] = \mathcal{F}_{\mathbf{1}}$.

Case T-THUNK $\qquad V = thunk\ C \qquad \dfrac{\Gamma \vdash_c C : T}{\Gamma \vdash_v thunk\ C : \mathcal{T}\, T}$

Now, $[\![thunk\ C]\!]\ \rho = [\![C]\!]\ \rho\ \in \mathcal{F}_T = \mathcal{F}_{\mathcal{T}\, T}$, by induction.

Case T-HEADER $\qquad V = h_i \qquad \dfrac{}{\Gamma \vdash_v h_i : \mathcal{H}}$

Immediately: $[\![h_i]\!]\ \rho = h_i \in \{h_i, \ldots, h_k\} = [\![\mathcal{H}]\!] = \mathcal{F}_{\mathcal{H}}$.

Case T-LIT $\qquad V = i \qquad \dfrac{}{\Gamma \vdash_v i : \mathbb{N}}$

Immediately: $[\![i]\!]\ \rho = i \in \mathbb{N} = [\![\mathbb{N}]\!] = \mathcal{F}_{\mathbb{N}}$.

Case Atomic Computations $C = P$ or $V_1 \leftarrow V_2$ or dup $\dfrac{\vdots}{\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}}$

By Lemma **??**, we know that $[\![C]\!]\ \rho\ A = \mathit{return}\ (\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\})$, then:

$$
\begin{aligned}
&\int g\ d([\![C]\!]\ \rho\ A) \\
&\quad \text{Lemma ??} \\
=&\int g\ d(\mathit{return}\ (\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\})) \\
&\quad \int g\ d(\mathit{return}\ x) = g(x) \\
=&\ g(\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\}) \\
&\quad \text{Since } \mathit{dom}(\lambda() \rightarrow \{f_c\ \rho\ x \mid x \in A\}) = \{()\},\ \chi_{F(\mathcal{P}\,\mathbf{1})}(\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\}) = 1 \\
=&\ \chi_{F(\mathcal{P}\,\mathbf{1})}(\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\})g((\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\}) \\
&\quad \int g\ d(\mathit{return}\ x) = g(x) \\
=&\int \chi_{F(\mathcal{P}\,\mathbf{1})}\ g\ d(\mathit{return}\ (\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\})) \\
&\quad \text{Lemma ??} \\
=&\int \chi_{F(\mathcal{P}\,\mathbf{1})}\ g\ d([\![C]\!]\ \rho\ A
\end{aligned}
$$

Hence, $[\![C]\!]\ \rho\ \in \mathcal{F}_{\mathcal{P}\,\mathbf{1}}$.

Case T-SEQ $\quad C = C_1 \,;C_2 \quad \dfrac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C : T_2}$

$$\int g \; d(\llbracket C_1 \,; C_2 \rrbracket \; \rho \; A)$$

by definition

$$= \int g \; d \left(\llbracket C_1 \rrbracket \; \rho \; A) \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket \; \rho \; \left(\bigcup_{x \in dom(\mu)} \mu(x) \right) \right)$$

$$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \;\; dm$$

$$= \int_\mu \int g \; d \left(\llbracket C_2 \rrbracket \; \rho \; \left(\bigcup_{x \in dom(\mu)} \mu(x) \right) \right) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

by induction for $\Gamma \vdash_c C_2 : T_2$

$$= \int_\mu \int \chi_{F(T_2)} \; g \; d \left(\llbracket C_2 \rrbracket \; \rho \; \left(\bigcup_{x \in dom(\mu)} \mu(x) \right) \right) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

$$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \;\; dm$$

$$= \int_\mu \int \chi_{F(T_2)} \; g \; d \left(\llbracket C_1 \rrbracket \; \rho \; A) \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket \; \rho \; \left(\bigcup_{x \in dom(\mu)} \mu(x) \right) \right)$$

by definition

$$= \int_\mu \int \chi_{F(T_2)} \; g \; d(\llbracket C_1 \,; C_2 \rrbracket \; \rho \; A)$$

Hence $\llbracket C_1 \,; C_2 \rrbracket \; \rho \; \in \mathcal{F}_{T_2}$.

Case T-Par $\qquad C = C_1 \,\&\, C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 :}$

$$\int g \; d(\llbracket C_1 \,\&\, C_2 \rrbracket \; \rho \; A)$$

by definition

$$= \int g \; d(\llbracket C_1 \rrbracket \; \rho \; A \gg\!= \lambda\mu_1 \to \llbracket C_2 \rrbracket \; \rho \; A \gg\!= \lambda\mu_2 \to \mathit{return} \; (\mu_1 \bigvee \mu_2))$$

$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \; dm$

$$= \int_{\mu_1} \int_{\mu_2} \int g \; d(\mathit{return} \; (\mu_1 \bigvee \mu_2)) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

$\int g \; d(\mathit{return} \; x) = g(x)$

$$= \int_{\mu_1} \int_{\mu_2} g(\mu_1 \bigvee \mu_2) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

by induction

$$= \int_{\mu_1} \chi_{F(T)}(\mu_1) \int_{\mu_2} \chi_{F(T)}(\mu_2) \; g(\mu_1 \bigvee \mu_2) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

distributivity

$$= \int_{\mu_1} \int_{\mu_2} \chi_{F(T)}(\mu_1) \; \chi_{F(T)}(\mu_2) \; g(\mu_1 \bigvee \mu_2) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

$\mathit{dom}(\mu \bigvee \mu_{n+1}) = \mathit{dom}(\mu_1) \cup \mathit{dom}(\mu_2)$,

then, $\mu_1 \bigvee \mu_2 \in F(T) \iff \mu, \mu_2 \in F(T)$

and therefore, $\chi_{F(T)}(\mu_1)\chi_{F(T)}(\mu_2) = \chi_{F(T)}(\mu_1 \bigvee \mu_2)$.

$$= \int_{\mu_1} \int_{\mu_2} \chi_{F(T)}(\mu_1 \bigvee \mu_2) \; g(\mu_1 \bigvee \mu_2) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

$\int g \; d(\mathit{return} \; x) = g(x)$

$$= \int_{\mu_1} \int_{\mu_2} \int \chi_{F(T)}(\mu_1 \bigvee \mu_2) \; g \; d(\mathit{return} \; (\mu_1 \bigvee \mu_2)) \; d(\llbracket C_2 \rrbracket \; \rho \; A) \; d(\llbracket C_1 \rrbracket \; \rho \; A)$$

$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \; dm$

$$= \int \chi_{F(T)}(\mu_1 \bigvee \mu_2) \; g \; d(\llbracket C_1 \rrbracket \; \rho \; A \gg\!= \lambda\mu_1 \to \llbracket C_2 \rrbracket \; \rho \; A \gg\!= \lambda\mu_2 \to \mathit{return} \; (\mu_1 \bigvee \mu_2))$$

by definition

$$= \int \chi_{F(T)}(\mu_1 \bigvee \mu_2) \; g \; d(\llbracket C_1 \,\&\, C_2 \rrbracket \; \rho \; A$$

Hence, $\llbracket C_1 \,\&\, C_2 \rrbracket \; \rho \; \in \mathcal{F}_{T_2}$.

Case T-Choice $C = C_1 \oplus_r C_2$ $$\frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus_r C_2 :}$$

$$\int g \; d(\llbracket C_1 \oplus_r C_2 \rrbracket \; \rho \; A)$$

by definition

$$= \int g \; d(r\llbracket C_1 \rrbracket \; \rho \; A + (1-r)\llbracket C_2 \rrbracket \; \rho \; A)$$

distributivity of addition, scalar multiplication

$$= r \int g \; d(\llbracket C_1 \rrbracket \; \rho \; A) + (1-r) \int g \; d(\llbracket C_2 \rrbracket \; \rho \; A))$$

by induction

$$= r \int \chi_{F(T)} \; g \; d(\llbracket C_1 \rrbracket \; \rho \; A) + (1-r) \int \chi_{F(T)} \; g \; d(\llbracket C_2 \rrbracket \; \rho \; A))$$

distributivity of addition, scalar multiplication

$$= \int \chi_{F(T)} \; g \; d(r\llbracket C_1 \rrbracket \; \rho \; A + (1-r)\llbracket C_2 \rrbracket \; \rho \; A)$$

by definition

$$= \int \chi_{F(T)} \; g \; d(r\llbracket C_1 \oplus_r C_2 \rrbracket \; \rho \; A)$$

Hence $\llbracket C_1 \oplus rC_2 \rrbracket \; \rho \; A \in \mathcal{F}_T$.

Case T-Iter $C = {C_1}^*$ $$\frac{\Gamma \vdash_c C_1 : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c {C_1}^* : \mathcal{P}\,\mathbf{1}}$$

$$\int g \; d(\llbracket {C_1}^* \rrbracket \; \rho \; A)$$

If $\mu : \llbracket \mathbf{1} \rrbracket \rightharpoonup A$, then $\mu : \mathcal{F}_{\mathbf{1}} \rightharpoonup A$ and $dom(\mu) = \{()\}$ is finite. Hence $\mu \in F(\mathcal{P}\,\mathbf{1})$.

$$= \int \chi_{F(\mathcal{P}\,\mathbf{1})} \; g \; d(\llbracket {C_1}^* \rrbracket \; \rho \; A$$

Hence $\llbracket {C_1}^* \rrbracket \; \rho \; A \in \mathcal{F}_{\mathcal{P}\,\mathbf{1}}$.

Case T-Produce $\quad C = \mathit{produce}\ V \quad \dfrac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \mathit{produce}\ V : \mathcal{P}\, S}$

$$\int g\ d(\llbracket \mathit{produce}\ V \rrbracket\ \rho\ A)$$

by definition

$$= \int g\ d(\mathit{return}\ (\lambda \llbracket V \rrbracket\ \rho \to A))$$

$$\int g\ d(\mathit{return}\ x) = g(x)$$

$$= g(\lambda \llbracket V \rrbracket\ \rho \to A)$$

By induction $\llbracket V \rrbracket\ \rho \in \mathcal{F}_S$, and $\mathit{dom}(\lambda \llbracket V \rrbracket\ \rho \to A) = \{\llbracket V \rrbracket\ \rho\}$ is finite.

Hence, $\lambda \llbracket V \rrbracket\ \rho \to A \in F(\mathcal{P}\, S)$.

$$= \chi_{F(\mathcal{P}\, S)}(\lambda \llbracket V \rrbracket\ \rho \to A)\ g(\lambda \llbracket V \rrbracket\ \rho \to A)$$

$$\int g\ d(\mathit{return}\ x) = g(x)$$

$$= \int \chi_{F(\mathcal{P}\, S)}\ g\ d(\mathit{return}\ (\lambda \llbracket V \rrbracket\ \rho \to A))$$

by definition

$$= \int \chi_{\mathcal{P}\, S}\ g\ d(\llbracket \mathit{produce}\ V \rrbracket\ \rho\ A)$$

Hence, $\llbracket \mathit{produce}\ V \rrbracket\ \rho\ \in \mathcal{F}_{\mathcal{P}\, S}$.

Case T-Force $\quad C = \mathit{force}\ V \quad \dfrac{\Gamma \vdash_v V : \mathcal{T}\, T}{\Gamma \vdash_c \mathit{force}\ V : T}$

$$\int g\ d(\llbracket \mathit{force}\ V \rrbracket\ \rho\ A)$$

by definition

$$= \int g\ d(\llbracket V \rrbracket\ \rho\ A)$$

$\Gamma \vdash_v V : \mathcal{T}\, T$, by inversion, $V = \mathit{thunk}\ C'$ where $\Gamma \vdash_c C' : T$.

$$= \int g\ d(\llbracket \mathit{thunk}\ C' \rrbracket\ \rho\ A)$$

by definition

$$= \int g\ d(\llbracket C' \rrbracket\ \rho\ A)$$

by induction

$$= \int \chi_{F(T)}\ g\ d(\llbracket C' \rrbracket\ \rho\ A)$$

by definition

$$= \int \chi_{F(T)}\ g\ d(\llbracket \mathit{force}\ V \rrbracket\ \rho\ A)$$

Hence, $[\![force\ V]\!]\ \rho\ \in \mathcal{F}_T$.

Case T-ABS $\quad C = \lambda x\!:\!S.C' \quad \dfrac{x : S, \Gamma \vdash_c C'\!:\!T}{\Gamma \vdash_c \lambda X\!:\!S.C'\!:\!S \to T}$

$$
\begin{aligned}
&\int g\ d([\![\lambda x\!:\!S.C']\!]\ \rho\ A)\\
&\quad \text{by definition}\\
&= \int g\ d(return\ (\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A))\\
&\quad \int g\ d(return\ x) = g(x)\\
&= g(\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A)\\
&\quad \text{If } v \in \mathcal{F}_S \text{, then } [\![C']\!]\ \rho\ \in \mathcal{F}_T \text{ by induction,}\\
&\quad \text{then, } \lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A \in \mathcal{F}_S \to 2^{PH} \to \mathcal{F}_T \text{, and}\\
&\quad \text{then, } dom(\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A) = \{\lambda v \to [\![C']\!]\ [x \mapsto v]\rho\} \text{ is finite.}\\
&= \chi_{F(S\to T)}(\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A)\ g(\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A)\\
&\quad \int g\ d(return\ x) = g(x)\\
&= \int \chi_{F(S\to T)}\ g\ d(return\ (\lambda(\lambda v \to [\![C']\!]\ [x \mapsto v]\rho) \to A))\\
&\quad \text{by definition}\\
&= \int \chi_{F(S\to T)}\ g\ d([\![\lambda x\!:\!S.C']\!]\ \rho\ A)
\end{aligned}
$$

Hence, $[\![\lambda x\!:\!S.C']\!]\ \rho\ \in \mathcal{F}_{S\to T}$.

Case T-App $\qquad C = C' \; V \qquad \dfrac{\Gamma \vdash_c C : S \to T \qquad \Gamma \vdash_v V : S}{\Gamma \vdash_c C' \; V : T}$

$$\int g \; d(\llbracket C' \; V \rrbracket \; \rho \; A)$$

by definition

$$= \int g \; d(\llbracket C' \rrbracket \; \rho \; A \gg\!= \lambda\mu \to \Xi\{f \; (\llbracket V \rrbracket \; \rho) \; \mu(v) \mid v \in dom(\mu)\})$$

$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \; dm$

$$= \int_\mu \int g \; d(\Xi\{f \; (\llbracket V \rrbracket \; \rho) \; \mu(v) \mid v \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A)$$

by induction

$$= \int_\mu \chi_{F(S \to T)}(\mu) \int \; d(\Xi\{f \; (\llbracket V \rrbracket \; \rho) \; \mu(v) \mid v \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A)$$

By induction $dom(\mu) \subseteq \mathcal{F}_{S \to T}$. Then the required follows from Lemma **??**.

$$= \int_\mu \int \chi_{F(T)} \; g \; d(\Xi\{f \; (\llbracket V \rrbracket \; \rho) \; \mu(v) \mid v \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A))$$

$\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \; dm$

$$= \int \chi_{F(T)} \; g \; d(\llbracket C' \rrbracket \; \rho \; A \gg\!= \lambda\mu \to \Xi\{f \; (\llbracket V \rrbracket \; \rho) \; \mu(v) \mid v \in dom(\mu)\})$$

by definition

$$= \int \chi_{F(T)} \; g \; d(\llbracket C' \; V \rrbracket \; \rho \; A)$$

Hence, $\llbracket C' \; V \rrbracket \; \rho \; \in \mathcal{F}_T$.

Case T-To $\quad C = C_1 \ \mathit{to}\ x.C_2 \quad \dfrac{\Gamma \vdash_c C_1 : \mathcal{P}\,S \qquad x : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \ \mathit{to}\ x.C_2 : T}$

$$\int g\ d(\llbracket C_1 \ \mathit{to}\ x.C_2 \rrbracket\ \rho\ A)$$

by definition

$$= \int g\ d(\llbracket C_1 \rrbracket\ \rho\ A \gg\!= \lambda\mu \to \Xi\{\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\})$$

$$\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$$

$$= \int_\mu \int g\ d(\Xi\{\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\})\ d(\llbracket C_1 \rrbracket\ \rho\ A)$$

by induction

$$= \int_\mu \chi_{F(\mathcal{P}\,S)}(\mu) \int g\ d(\Xi\{\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\})\ d(\llbracket C_1 \rrbracket\ \rho\ A)$$

Since $\mu \in F(\mathcal{P}\,S)$, it follows that $\forall v \in \mathit{dom}(\mu) : v \in \mathcal{F}_S$, then, by induction, $\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \in \mathcal{F}_T$.

$$= \int_\mu \int \chi_{F(T)}\ g\ d(\Xi\{\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\})\ d(\llbracket C_1 \rrbracket\ \rho\ A)$$

$$\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$$

$$= \int \chi_{F(T)}\ g\ d(\llbracket C_1 \rrbracket\ \rho\ A \gg\!= \lambda\mu \to \Xi\{\llbracket C_2 \rrbracket\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\})$$

by definition

$$= \int \chi_{F(T)}\ g\ d(\llbracket C_1 \ \mathit{to}\ x.C_2 \rrbracket\ \rho\ A)$$

Hence, $\llbracket C_1 \ \mathit{to}\ x.C_2 \rrbracket\ \rho\ \in \mathcal{F}_T$. □

Lemma C.1. *Let $M = \{m_1, \ldots, m_n\} \subseteq \llbracket T \rrbracket$, such that $\int f\ dm_i = \int \chi_{F(T)}\ f\ dm_i$, for $i = 1, \ldots, n$; then*

$$\int f\ d(\Xi M) = \int \chi_{F(T)}\ f d(\Xi M)$$

Proof. By induction on n:

- $n = 0$

$$
\begin{aligned}
& \int g \; d(\Xi \emptyset) \\
& \quad \text{by definition, } \bigvee \emptyset = \lambda x \to \bot \\
= & \int g \; d(\mathit{return}\ (\lambda x \to \bot)) \\
& \quad \int g \; d(\mathit{return}\ x) = g(x) \\
= & g(\lambda x \to \bot) \\
& \quad \mathit{dom}(()\lambda x \to \bot) \text{ is finite} \\
= & \chi_{F(T)}(\lambda x \to \bot) g(\lambda x \to \bot) \\
& \quad \int g \; d(\mathit{return}\ x) = g(x) \\
= & \int \chi_{F(T)} \; g \; d(\mathit{return}\ (\lambda x \to \bot)) \\
& \quad \text{by definition, } \bigvee \emptyset = \lambda x \to \bot \\
= & \int \chi_{F(T)} \; g \; d(\Xi \emptyset)
\end{aligned}
$$

- Induction

$$\int g\ d(\Xi\{m_1,\ldots,m_n,m_{n+1}\})$$

by definition

$$= \int g\ d(\Xi\{m_1,\ldots,m_n\} \gg\!= \lambda\mu \to m_{n+1} \gg\!= \lambda\mu_{n+1} \to \mathit{return}\ (\mu \bigvee \mu_{n+1}))$$

$$\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$$

$$= \int_\mu \int_{\mu_{n+1}} \int g\ d(\mathit{return}\ (\mu \bigvee \mu_{n+1}))\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

$$\int g\ d(\mathit{return}\ x) = g(x)$$

$$= \int_\mu \int_{\mu_{n+1}} g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

by induction and by assumption for m_{n+1}

$$= \int_\mu \chi_{F(T)}(\mu) \int_{\mu_{n+1}} \chi_{F(T)}(\mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

distributivity

$$= \int_\mu \int_{\mu_{n+1}} \chi_{F(T)}(\mu)\ \chi_{F(T)}(\mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

$\mathit{dom}(\mu \bigvee \mu_{n+1}) = \mathit{dom}(\mu) \cup \mathit{dom}(\mu_{n+1})$,

then, $\mu \bigvee \mu_{n+1} \in F(T) \iff \mu, \mu_{n+1} \in F(T)$

and therefore, $\chi_{F(T)}(\mu)\chi_{F(T)}(\mu_{n+1}) = \chi_{F(T)}(\mu \bigvee \mu_{n+1})$.

$$= \int_\mu \int_{\mu_{n+1}} \chi_{F(T)}(\mu \bigvee \mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

$$= \int \chi_{F(T)}\ g\ d(\Xi\{m_1,\ldots,m_n,m_{n+1}\}$$

□

C.3 Theorem 5.7

Proof. By induction on the structure of the typing derivation, performing case analysis on the final rule application.

In what follows, let $\Gamma \vdash_v V : S$, $\Gamma \vdash_c C : T$, and $\rho \in [\![\Gamma]\!]$, such that $\forall (x : S_x) \in \Gamma : \rho(x) \in \mathcal{C}_{S_x}$.

Case T-Var $\quad V = x_i \quad \dfrac{x_i : S_i \in \Gamma}{\Gamma \vdash_v x_i : S_i}$

By assumption, $[\![x_i]\!]\ \rho = \rho(x_i) \in \mathcal{C}_{S_i}$.

Case T-Unit $\quad V = \mathit{unit} \quad \Gamma \vdash_v \mathit{unit} : \mathbf{1}$

$$[\![\mathit{unit}]\!]\ \rho = () \in \{()\} = [\![\mathbf{1}]\!] = \mathcal{C}_{\mathbf{1}}$$

Case T-HEADER $\quad V = h_i \quad \Gamma \vdash_v h_i : \mathcal{H}$

$$[\![h_i]\!]\ \rho = h_i \in \{h_1, \ldots, h_n\} \in [\![\mathcal{H}]\!] = \mathcal{C}_{\mathcal{H}}$$

Case T-LIT $\quad V = n \quad \Gamma \vdash_v n : \mathbb{N}$

$$[\![n]\!]\ \rho = n \in \mathbb{N} = [\![\mathbb{N}]\!] = \mathcal{C}_{\mathbb{N}}$$

Case T-THUNK $\quad V = \mathit{thunk}\ C \quad \dfrac{\Gamma \vdash_c C : T}{\Gamma \vdash_v \mathit{thunk}\ C : \mathcal{T}\,T}$

Case ATOMIC COMPUTATIONS $\quad C = P$ or $V_1 \leftarrow V_2$ or $\mathit{dup} \quad \dfrac{\vdots}{\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}}$

By Lemma **??**, we know that $[\![C]\!]\ \rho\ A = \mathit{return}\ (\lambda() \rightarrow \{f_C\ \rho\ x \mid x \in A\})$, then:

(1) We show that $[\![C]\!]\ \rho$ is continuous:

$$\begin{aligned}
&\bigsqcup_{i \geq 0} [\![C]\!]\ \rho\ A_i \\
=& \bigsqcup_{i \geq 0} (\mathit{return}\ (\lambda() \rightarrow \{f_c(x) \mid x \in A_i\})) \\
=& \mathit{return}\ \left(\bigvee_{i \geq 0} (\lambda() \rightarrow \{f_c(x) \mid x \in A_i\}) \right) \\
=& \mathit{return}\ \left(\lambda() \rightarrow \bigcup_{i \geq 0} \{f_c(x) \mid x \in A_i\} \right) \\
=& \mathit{return}\ \left(\lambda() \rightarrow \{f_c(x) \mid x \in \bigcup_{i \geq 0} A_i\} \right) \\
=& [\![C]\!]\ \rho\ \left(\bigcup_{i \geq 0} A_i \right)
\end{aligned}$$

(2)

$$
\begin{aligned}
&\int g\, d\,(\llbracket C \rrbracket\ \rho\ A)\\
&\quad \text{by definition}\\
&=\int g\, d\,(\mathit{return}\ (\lambda() \to \{f_C(x) \mid x \in A\}))\\
&\quad \int g\ d(\mathit{return}\ x) = g(x)\\
&=g(\lambda() \to \{f_C(x) \mid x \in A\})\\
&\quad \mathit{dom}(\lambda() \to \{f_C(x) \mid x \in A\}) = \mathcal{C}_(\mathbf{1})\\
&\quad \text{Thus, } \lambda() \to \{f_C(x) \mid x \in A\} \in C_{\mathcal{P}\,\mathbf{1}}\\
&=\chi_{C(\mathcal{P}\,\mathbf{1})}(\lambda() \to \{f_C(x) \mid x \in A\})\ g(\lambda() \to \{f_C(x) \mid x \in A\})\\
&\quad \int g\ d(\mathit{return}\ x) = g(x)\\
&=\int \chi_{C(\mathcal{P}\,\mathbf{1})} g\ d\,(\mathit{return}\ (\lambda() \to \{f_C(x) \mid x \in A\}))\\
&\quad \text{by definition}\\
&=\int \chi_{C(\mathcal{P}\,\mathbf{1})} g\ d\,(\llbracket C \rrbracket\ \rho\ A)
\end{aligned}
$$

Thus $\llbracket C \rrbracket\ \rho\ \in \mathcal{C}_{\mathcal{P}\,\mathbf{1}}$.

Case T-SEQ $\qquad C = C_1\,;C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C : T_2}$

(1) We show that $\llbracket C_1\,;C_2 \rrbracket\ \rho\ A$ is continuous:

$$
\begin{aligned}
&\bigsqcup_{i\geq 0} \llbracket C_1\,;C_2 \rrbracket\ \rho\ A_i\\
&=\bigsqcup_{i\geq 0}\left(\llbracket C_1 \rrbracket\ \rho\ A_i \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket\ \rho\ \Big(\bigcup_{x\in \mathit{dom}(\mu)} \mu(x)\Big)\right)\\
&\quad \text{continuity of } \gg\!=\text{, assume } \lambda\mu \to \llbracket C_2 \rrbracket\ \rho\ \Big(\bigcup_{x\in \mathit{dom} \mu} \mu(x)\Big) \text{ is continuous}\\
&=\bigsqcup_{i\geq 0} (\llbracket C_1 \rrbracket\ \rho\ A_i) \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket\ \rho\ \Big(\bigcup_{x\in \mathit{dom}(\mu)} \mu(x)\Big)\\
&=\llbracket C_1 \rrbracket\ \rho\ \Big(\bigcup_{i\geq 0} A_i\Big) \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket\ \rho\ \Big(\bigcup_{x\in \mathit{dom}(\mu)} \mu(x)\Big)\\
&=\llbracket C_1\,;C_2 \rrbracket\ \rho\ \Big(\bigcup_{i\geq 0} A_i\Big)
\end{aligned}
$$

Moreover,

$$
\begin{aligned}
&\bigsqcup_{i\geq 0} [\![C_2]\!]\ \rho\ \big(\bigcup_{x\in dom(\mu_i)} \mu_i(x)\big)\\
=&[\![C_2]\!]\ \rho\ \big(\bigcup_{\substack{i\geq 0\\ x\in dom(\mu_i)}} \mu_i(x)\big)\\
=&[\![C_2]\!]\ \rho\ \big(\bigcup_{\substack{i\geq 0\\ x\in dom(\bigvee_{j\geq 0}\mu_j)}} \mu_i(x)\big)\\
=&[\![C_2]\!]\ \rho\ \big(\bigcup_{x\in dom(\bigvee_{j\geq 0}\mu_j)} (\bigvee_{j\geq 0} \mu_j)(x)\big)
\end{aligned}
$$

(2)

$$
\begin{aligned}
&\int g\ d([\![C_1\,;C_2]\!]\ \rho\ A)\\
=&\int g\ d([\![C_1]\!]\ \rho\ A \ggg\!= \lambda\mu \to [\![C_2]\!]\ \rho \bigcup_{x\in dom(\mu)} \mu(x))\\
=&\int_\mu \int g\ d([\![C_2]\!]\ \rho \bigcup_{x\in dom(\mu)} \mu(x))\ d([\![C_1]\!]\ \rho\ A\\
&\text{By induction on } \Gamma \vdash_c C_2 : T_2.\\
=&\int_\mu \int \chi_{\mathcal{C}_{T_2}} g\ d([\![C_2]\!]\ \rho \bigcup_{x\in dom(\mu)} \mu(x))\ d([\![C_1]\!]\ \rho\ A\\
=&\int \chi_{\mathcal{C}_{T_2}} g; d([\![C_1]\!]\ \rho\ A \ggg\!= \lambda\mu \to [\![C_2]\!]\ \rho \bigcup_{x\in dom(\mu)} \mu(x))\\
&\int \chi_{\mathcal{C}_{T_2}} g\ d([\![C_1\,;C_2]\!]\ \rho\ A)
\end{aligned}
$$

Thus $[\![C_1\,;C_2]\!]\ \rho \in \mathcal{C}_{T_2}$.

Case T-Par $\qquad C = C_1 \,\&\, C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 :}$

(1) We show that $[\![C_1 \& C_2]\!]\ \rho$ is continuous:

$$
\begin{aligned}
&\bigsqcup_{i\geq 0} [\![C_1 \& C_2]\!]\ \rho\ A_i \\
=&\bigsqcup_{i\geq 0}\Big([\![C_1]\!]\ \rho\ A_i \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A_i \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2)\Big) \\
&\text{Continuity of } \gg\!= \text{ and } \mathit{return}\ (\mu_1 \bigvee \mu_2) \text{ is continuous} \\
=&\bigsqcup_{i\geq 0}([\![C_1]\!]\ \rho\ A_i) \gg\!= \lambda\mu_1 \to \bigsqcup_{i\geq 0}([\![C_2]\!]\ \rho\ A_i) \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
=&[\![C_1]\!]\ \rho\ (\bigcup_{i\geq 0} A_i) \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i) \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
=&[\![C_1 \& C_2]\!]\ \rho\ (\bigcup_{i\geq 0})
\end{aligned}
$$

(2)

$$
\begin{aligned}
&\int g\ d([\![C_1 \& C_2]\!]\ \rho\ A) \\
=&\int g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
=&\int_{\mu_1}\int_{\mu_2} g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
=&\int_{\mu_1} \chi_{C(T)}(\mu_1) \int_{\mu_2} \chi_{C(T)}(\mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
&\text{Induction on } \Gamma \vdash_c C_1 : T, \Gamma \vdash_c C_2 : T \\
=&\int_{\mu_1}\int_{\mu_2} \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
&\text{If } \mu_1, \mu_2 \in C(T), \text{ then } \mu_1 \bigvee \mu_2 \in C(T) \\
&\text{Hence } \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2) = \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2) \\
=&\int_{\mu_1}\int_{\mu_2} \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2)\chi_{C(T)}(\mu_1 \bigvee \mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
&\text{Induction on } \Gamma \vdash_c C_1 : T, \Gamma \vdash_c C_2 : T \\
=&\int_{\mu_1}\int_{\mu_2} \chi_{C(T)}(\mu_1 \bigvee \mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
=&\int \chi_{C(T)} g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2)) \\
=&\int \chi_{C(T)} g\ d([\![C_1 \& C_2]\!]\ \rho\ A)
\end{aligned}
$$

Thus $[\![C_1 C \oplus_2]\!]\ \rho \in \mathcal{C}_T$.

Case T-CHOICE $\quad C = C_1 \oplus_r C_2 \quad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus_r C_2 :}$

(1) We show that $[\![C_1 \oplus C_2]\!]\ \rho$ is continuous.

$$\begin{aligned}
&\bigsqcup_{i\geq 0} [\![C_1 \oplus C_2]\!]\ \rho\ A_i \\
=&\bigsqcup_{i\geq 0} (r([\![C_1]\!]\ \rho\ A_i) + (1-r)([\![C_2]\!]\ \rho\ A_i)) \\
=&\, r\bigsqcup_{i\geq 0} ([\![C_1]\!]\ \rho\ A_i) + (1-r)\bigsqcup_{i\geq 0} ([\![C_2]\!]\ \rho\ A_i)) \\
=&\, r([\![C_1]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)) + (1-r)([\![C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)) \\
=&[\![C_1 \oplus C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)
\end{aligned}$$

(2)

$$\begin{aligned}
&\int g\ d([\![C_1 \oplus_r C_2]\!]\ \rho\ A) \\
=&\, r\int g\ d([\![C_1]\!]\ \rho\ A) + (1-r)\int g\ d([\![C_2]\!]\ \rho\ A) \\
&\quad \text{by induction} \\
=&\, r\int \chi_{C(T)} g\ d([\![C_1]\!]\ \rho\ A) + (1-r)\int \chi_{C(T)} g\ d([\![C_2]\!]\ \rho\ A) \\
=&\int \chi_{C(T)} g\ d\big(r([\![C_1]\!]\ \rho\ A) + (1-r)([\![C_2]\!]\ \rho\ A)\big) \\
=&\int \chi_{C(T)} g\ d\big([\![C_1 \oplus_r C_2]\!]\ \rho\ A\big)
\end{aligned}$$

Thus, $[\![C_1 \oplus C_2]\!]\ \rho \in \mathcal{C}_T$.

Case T-ITER $\qquad C = {C_1}^* \qquad \dfrac{\Gamma \vdash_c C_1 : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c {C_1}^* : \mathcal{P}\,\mathbf{1}}$

(1) We show that $[\![{C_1}^*]\!]\ \rho$ is continuous:

$$\begin{aligned}
&\bigsqcup_{i\geq 0} ([\![{C_1}^*]\!]\ \rho\ A_i) \\
=&\bigsqcup_{i\geq 0} (\bigsqcup_{n\geq 0} [\![C_1^n]\!]\ \rho\ A_i) \\
=&\bigsqcup_{n\geq 0} [\![C_1^n]\!]\ \rho\ (\bigcup_{i\geq 0} A_i) \\
=&[\![{C_1}^*]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)
\end{aligned}$$

(2)

$$
\begin{aligned}
&\int g \; d(\llbracket {C_1}^* \rrbracket \; \rho \; A) \\
=&\int g \; d(\bigsqcup_{n\geq 0} \llbracket C_1^n \rrbracket \; \rho \; A) \\
&\quad \color{green}{\mathcal{C}_{\mathbf{1}} = \llbracket \mathbf{1} \rrbracket = \{()\}, \text{ then clearly } \mu \in \llbracket \mathbf{1} \rrbracket \rightharpoonup 2^{PH} = C(\mathcal{P}\,\mathbf{1})} \\
=&\int \chi_{C(\mathcal{P}\,\mathbf{1})} g \; d(\bigsqcup_{n\geq 0} \llbracket C_1^n \rrbracket \; \rho \; A) \\
=&\int \chi_{C(\mathcal{P}\,\mathbf{1})} g \; d(\llbracket {C_1}^* \rrbracket \; \rho \; A)
\end{aligned}
$$

Thus $\llbracket C \rrbracket \; \rho \; \in \mathcal{C}_{\mathcal{P}\,\mathbf{1}}$.

Case T-Produce $\quad C = \mathit{produce}\ V \quad \dfrac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \mathit{produce}\ V : \mathcal{P}\, S}$

(1) We show that $\llbracket \mathit{produce}\ V \rrbracket \; \rho$ is continuous.

$$
\begin{aligned}
&\bigsqcup_{i\geq 0} \llbracket \mathit{produce}\ V \rrbracket \; \rho \; A_i \\
=&\bigsqcup_{i\geq 0} \mathit{return}\ (\lambda \llbracket V \rrbracket \; \rho \to A_i) \\
=&\mathit{return}\ \bigvee_{i\geq 0} (\lambda \llbracket V \rrbracket \; \rho \to A_i) \\
=&\mathit{return}\ (\lambda \llbracket V \rrbracket \; \rho \to \bigcup_{i\geq 0} A_i) \\
=&\llbracket \mathit{produce}\ V \rrbracket \; \rho \; (\bigcup_{i\geq 0} A_i)
\end{aligned}
$$

(2)

$$
\begin{aligned}
&\int g \; d(\llbracket \mathit{produce}\ V \rrbracket \; \rho \; A \\
=&\int g \; d(\mathit{return}\ (\lambda \llbracket V \rrbracket \; \rho \to A)) \\
=&g(\lambda \llbracket V \rrbracket \; \rho \to A) \\
&\quad \color{green}{\Gamma \vdash_c V : S, \text{ by induction } V \in \mathcal{C}_S} \\
=&\chi_{C(\mathcal{P}\,S)}(\lambda \llbracket V \rrbracket \; \rho \to A) g(\lambda \llbracket V \rrbracket \; \rho \to A) \\
=&\int \chi_{C(\mathcal{P}\,S)} g \; d(\mathit{return}\ (\lambda \llbracket V \rrbracket \; \rho \to A)) \\
=&\int \chi_{C(\mathcal{P}\,S)} g \; d(\llbracket \mathit{produce}\ V \rrbracket \; \rho \; A)
\end{aligned}
$$

Thus $\llbracket \mathit{produce}\ V \rrbracket\ \rho \in \mathcal{C}_{\mathcal{P}\ S}$.

Case T-Force $\quad C = \mathit{force}\ V \quad \dfrac{\Gamma \vdash_v V : \mathcal{T}\,T}{\Gamma \vdash_c \mathit{force}\ V : T}$

$$\llbracket \mathit{force}\ V \rrbracket\ \rho = \llbracket V \rrbracket\ \rho \in \mathcal{C}_{\mathcal{T}\ T} = \mathcal{C}_T$$

where the first and last equality are by definition, the middle one is by induction.

Case T-Abs $\quad C = \lambda x : S.C' \quad \dfrac{x : S, \Gamma \vdash_c C' : T}{\Gamma \vdash_c \lambda X : S.C' : S \to T}$

(1) We show that $\llbracket \lambda x : S.C' \rrbracket\ \rho$ is continuous:

$$\begin{aligned}
&\bigsqcup_{i \geq 0} \llbracket \lambda x : S.C \rrbracket\ \rho\ A_i \\
=&\bigsqcup_{i \geq 0} \mathit{return}\ (\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \to A_i) \\
=&\mathit{return}\ \bigvee_{i \geq 0} (\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \to A_i) \\
=&\mathit{return}\ ((\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \to \bigcup_{i \geq 0} A_i) \\
=&\llbracket \lambda x : S.C \rrbracket\ \rho\ (\bigcup_{i \geq 0} A_i)
\end{aligned}$$

(2)

$$\begin{aligned}
&\int g\ d(\llbracket \lambda x : S.C' \rrbracket\ \rho\ A) \\
=&\int g\ d(\mathit{return}\ \lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \to A) \\
=&g(\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho\ ) \to A) \\
&\quad \text{If } v \in \mathcal{C}_S \text{, then } \llbracket C' \rrbracket\ [x \mapsto v]\rho\ \in \mathcal{C}_T \text{, by induction.} \\
&\quad \text{Hence, } (\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \in \mathcal{C}_S \to \mathcal{C}_T \\
=&\chi_{C(S \to T)} g(\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho\ ) \to A) g(\lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho\ ) \to A) \\
=&\int \chi_{C(S \to T)} g\ d(\mathit{return}\ \lambda(\lambda v \to \llbracket C' \rrbracket\ [x \mapsto v]\rho) \to A) \\
=&\int \chi_{C(S \to T)} g\ d(\llbracket \lambda x : S.C' \rrbracket\ )
\end{aligned}$$

Thus, $\llbracket \lambda x : S.C' \rrbracket\ \rho \in \mathcal{C}_{S \to T}$.

Case T-App $\quad C = C'\ V \quad \dfrac{\Gamma \vdash_c C : S \to T \qquad \Gamma \vdash_v V : S}{\Gamma \vdash_c C'\ V : T}$

(1) We show that $[\![C'\ V]\!]\ \rho$ is continuous:

$$\begin{aligned}
&\bigsqcup_{i\geq 0}[\![C'\ V]\!]\ \rho\ A_i\\
=&\bigsqcup_{i\geq 0}([\![C']\!]\ \rho\ A_i \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho)\ \mu(f)\mid f\in dom(\mu)\})\\
&\text{continuity of } \gg\!=\text{, assume } \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho)\ \mu(f)\mid f\in dom(\mu)\} \text{ is continuous}\\
=&\bigsqcup_{i\geq 0}([\![C']\!]\ \rho\ A_i) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho)\ \mu(f)\mid f\in dom(\mu)\})\\
=&[\![C']\!]\ \rho\ \big(\bigcup_{i\geq 0} A_i\big) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho)\ \mu(f)\mid f\in dom(\mu)\})\\
=&[\![C'\ V]\!]\ \rho\ \big(\bigcup_{i\geq 0} A_i\big)
\end{aligned}$$

Moreover,

$$\begin{aligned}
&\bigsqcup_{i\geq 0}\Xi\{f\ ([\![V]\!]\ \rho)\ \mu_i(f)\mid f\in dom(\mu_i)\}\\
=&\bigsqcup_{\substack{i\geq 0\\ f\in dom(\mu_i)}}(f\ ([\![V]\!]\ )\rho)\ \mu_i(f))\\
=&\bigsqcup_{\substack{i\geq 0\\ f\in dom(\bigvee_{j\geq 0}\mu_j)}}(f\ ([\![V]\!]\ )\rho)\ \mu_i(f))\\
=&\bigsqcup_{f\in dom(\bigvee_{i\geq 0}\mu_i)}\left(f\ ([\![V]\!]\ )\rho)\ (\bigcup_{i\geq 0}\mu_i(f))\right)\\
=&\bigsqcup_{f\in dom(\bigvee_{i\geq 0}\mu_i)}\left(f\ ([\![V]\!]\ )\rho)\ (\bigvee_{i\geq 0}\mu_i)(f)\right)\\
=&\ \Xi\{f\ ([\![V]\!]\ \rho)\ (\bigvee_{i\geq 0}\mu_i)\mid f\in dom(\bigvee_{i\geq 0}\mu_i)\}
\end{aligned}$$

Discharging the assumption above.

(2)

$$
\begin{aligned}
& \int g \; d(\llbracket C' \; V \rrbracket \; \rho \; A) \\
=& \int g \; d(\llbracket C' \rrbracket \; \rho \; A \gg\!= \lambda \mu \to \Xi\{ f \; (\llbracket V \rrbracket \; \rho) \; \mu(f) \mid f \in dom(\mu)\}) \\
=& \int_\mu \int g \; d(\Xi\{ f \; (\llbracket V \rrbracket \; \rho) \; \mu(f) \mid f \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A) \\
& \quad \text{by induction} \\
=& \int_\mu \chi_{C(S \to T)}(\mu) \int g \; d(\Xi\{ f \; (\llbracket V \rrbracket \; \rho) \; \mu(f) \mid f \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A) \\
& \quad \mu \in C(S \to T) \text{, then } f \; \llbracket V \rrbracket \; \rho \in \mathcal{C}_T \text{, thus } \int g \; d(f \; \llbracket V \rrbracket \; \rho A) = \int \chi_{C(T)} g \; d(f \; \llbracket V \rrbracket \; \rho A) \\
& \quad \text{Now apply Lemma \textbf{??}.} \\
=& \int_\mu \chi_{C(S \to T)}(\mu) \int \chi_{C(T)} g \; d(\Xi\{ f \; (\llbracket V \rrbracket \; \rho) \; \mu(f) \mid f \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A) \\
=& \int_\mu \int \chi_{C(T)} g \; d(\Xi\{ f \; (\llbracket V \rrbracket \; \rho) \; \mu(f) \mid f \in dom(\mu)\}) \; d(\llbracket C' \rrbracket \; \rho \; A) \\
=& \int \chi_{C(T)} g d(\llbracket C' \; V \rrbracket \; \rho \; A)
\end{aligned}
$$

Thus, $\llbracket C' \; V \rrbracket \; \rho \in \mathcal{C}_T$.

Case T-To $\qquad C = C_1 \; to \; x.C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : \mathcal{P}\, S \qquad x : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \; to \; x.C_2 : T}$

(1) We show that $\llbracket C_1 \; to \; x.C_2 \rrbracket \; \rho$ is continuous:

$$
\begin{aligned}
& \bigsqcup_{i \geq 0} (\llbracket C_1 \; to \; x.C_2 \rrbracket \; \rho \; A_i) \\
=& \bigsqcup_{i \geq 0} (\llbracket C_1 \rrbracket \; \rho \; A_i \gg\!= \lambda \mu \to \Xi\{ \llbracket C_2 \rrbracket \; [x \mapsto v]\rho \; \mu(v) \mid v \in dom(\mu)\}) \\
& \quad \text{continuity of } \gg\!=\text{, assume } \lambda \mu \to \Xi\{ \llbracket C_2 \rrbracket \; [x \mapsto v]\rho \; \mu(v) \mid v \in dom(\mu)\} \text{ is continuous.} \\
=& \bigsqcup_{i \geq 0} (\llbracket C_1 \rrbracket \; \rho \; A_i) \gg\!= \lambda \mu \to \Xi\{ \llbracket C_2 \rrbracket \; [x \mapsto v]\rho \; \mu(v) \mid v \in dom(\mu)\} \\
& \quad \text{by induction} \\
=& \llbracket C_1 \rrbracket \; \rho \; (\bigsqcup_{i \geq 0} A_i) \gg\!= \lambda \mu \to \Xi\{ \llbracket C_2 \rrbracket \; [x \mapsto v]\rho \; \mu(v) \mid v \in dom(\mu)\}
\end{aligned}
$$

Moreover,

$$\begin{aligned}
&\bigsqcup_{i\geq 0} \Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu_i(v) \mid v\in dom(\mu_i)\}\\
=&\bigsqcup_{\substack{i\geq 0\\ f\in dom(\mu_i)}} [\![C_2]\!]\ [x\mapsto v]\rho\ \mu_i(v)\\
=&\bigsqcup_{\substack{i\geq 0\\ f\in dom(\bigvee_{j\geq 0}\mu_j)}} [\![C_2]\!]\ [x\mapsto v]\rho\ \mu_i(v)\\
=&\bigsqcup_{f\in dom(\bigvee_{j\geq 0}\mu_j)} [\![C_2]\!]\ [x\mapsto v]\rho\ \big(\bigcup_{i\geq 0}\mu_i(v)\big)\\
=&\bigsqcup_{f\in dom(\bigvee_{j\geq 0}\mu_j)} [\![C_2]\!]\ [x\mapsto v]\rho\ \big(\bigvee_{i\geq 0}\mu_i\big)(v)\\
=&\Xi\left\{[\![C_2]\!]\ [x\mapsto v]\rho\ \big(\bigvee_{i\geq 0}\mu_i\big)(v) \mid v\in dom(\big(\bigvee_{i\geq 0}\mu_i\big))\right\}
\end{aligned}$$

This discharges the above assumption.

(2)

$$\begin{aligned}
&\int g\ d([\![C_1\ to\ x.C_2]\!]\ \rho\ A)\\
=&\int g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu(v) \mid v\in dom(\mu)\})\\
=&\int_\mu \int g\ d(\Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu(v) \mid v\in dom(\mu)\})\ d([\![C_1]\!]\ \rho\ A)
\end{aligned}$$

by induction

$$= \int_\mu \chi_{C(\mathcal{P}\ S)}(\mu) \int g\ d(\Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu(v) \mid v\in dom(\mu)\})\ d([\![C_1]\!]\ \rho\ A)$$

By induction $[\![C_2]\!]\ [x\mapsto v]\rho \in \mathcal{C}_T$, thus $\int g\ d([\![C_2]\!]\ [x\mapsto v]\rho\ A) = \int \chi_{C(T)} g\ d([\![C_2]\!]\ [x\mapsto v]\rho\ A)$

Now apply Lemma **??**.

$$\begin{aligned}
=&\int_\mu \chi_{C(\mathcal{P}\ S)}(\mu) \int \chi_{C(T)} g\ d(\Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu(v) \mid v\in dom(\mu)\})\ d([\![C_1]\!]\ \rho\ A)\\
=&\int_\mu \int \chi_{C(T)} g\ d(\Xi\{[\![C_2]\!]\ [x\mapsto v]\rho\ \mu(v) \mid v\in dom(\mu)\})\ d([\![C_1]\!]\ \rho\ A)
\end{aligned}$$

Thus $[\![C_1\ to\ x.C_2]\!]\ \rho \in \mathcal{C}_T$. □

Lemma C.2. *Let* $M = \{m_1,\dots,m_n\} \subseteq [\![T]\!]$, *such that* $\int f\ dm_i = \int \chi_{C(T)}\ f\ dm_i$, *for* $i = 1,\dots,n$; *then*

$$\int f\ d(\Xi M) = \int \chi_{C(T)}\ f d(\Xi M)$$

Proof.

$$
\begin{aligned}
&\Xi M \\
&= \int_{\mu_1} \cdots \int_{\mu_n} \int g \; d(\mathit{return}\ (\mu_1 \bigvee \cdots \mu_n))\ dm_n \cdots dm_1 \\
&= \int_{\mu_1} \chi_{C(T)}(\mu_1) \cdots \int_{\mu_n} \chi_{C(T)}(\mu_n) \int g \; d(\mathit{return}\ (\bigvee_{i=1}^{n} \mu_i))\ dm_n \cdots dm_1 \\
&\quad \text{If } \mu_1, \ldots, \mu_n \in C(T), \text{ then } \mu_1 \bigvee \cdots \bigvee \mu_n \in C(T). \\
&\quad \text{Since } \mathit{dom}(\mu_1 \bigvee \cdots \bigvee \mu_n) = \mathit{dom}(\mu_1) \cup \cdots \cup \mathit{dom}(\mu_n) \subseteq \mathcal{C}_S \text{ or } \mathcal{C}_S \to \mathcal{C}_T. \\
&= \int_{\mu_1} \chi_{C(T)}(\mu_1) \cdots \int_{\mu_n} \chi_{C(T)}(\mu_n) \int \chi_{C(T)}\ g \; d(\mathit{return}\ (\bigvee_{i=1}^{n} \mu_i))\ dm_n \cdots dm_1 \\
&= \int_{\mu_1} \cdots \int_{\mu_n} \int \chi_{C(T)}\ g \; d(\mathit{return}\ (\bigvee_{i=1}^{n} \mu_i))\ dm_n \cdots dm_1
\end{aligned}
$$

□

C.4 Theorem 5.8

Recall the definition of PNK semantics given in Appendix **??**. To prove Theorem 5.8, we actually prove the following lemmas:

Lemma C.3. *Let P be a closed predicate. Then for all $A \in 2^{PH}$:*

$$\mathit{return}\ (\llbracket P \rrbracket^p\ ()\ A) = \llbracket P \rrbracket_{PNK}\ A$$

Proof. By induction on the structure of predicates:

Cases drop, skip, tests $f = n$. : straightforward by definition.

Case $\neg P$.

$$
\begin{aligned}
\mathit{return}\ (\llbracket \neg P \rrbracket^p\ ()\ A) &= \mathit{return}\ (A - \llbracket P \rrbracket^p\ ()\ A) \\
&= \mathit{return}\ (\llbracket P \rrbracket^p\ ()\ A) \gg\!= \lambda B \to \mathit{return}\ (A - B) \\
&= \llbracket P \rrbracket_{PNK}\ A \gg\!= \lambda B \to \mathit{return}\ (A - B) \\
&= \llbracket \neg P \rrbracket_{PNK}\ A
\end{aligned}
$$

Case $P_1 \wedge P_2$.

$$
\begin{aligned}
\mathit{return}\ (\llbracket P_1 \wedge P_2 \rrbracket^p\ ()\ A) &= \mathit{return}\ \{h \in A \mid B_{P_1 \wedge P_2}\ ()\ h\} \\
&= \mathit{return}\ \{h \in A \mid B_{P_1}\ ()\ h \text{ and } B_{P_2}\ ()\ h\} \\
&= \mathit{return}\ \{h \in \{h \in A \mid B_{P_1}\ ()\ h\} \mid B_{P_2}\ ()\ h\} \\
&= \mathit{return}\ \{h \in A \mid B_{P_1}\ ()\ h\} \gg\!= \lambda A' \to \mathit{return}\ \{h \in A' \mid B_{P_2}\ ()\ h\} \\
&= \llbracket P_1 \rrbracket_{PNK}\ A \gg\!= \lambda A' \to \llbracket P_2 \rrbracket_{PNK}\ A' \\
&= \llbracket P_1 \rrbracket_{PNK}\ A \gg\!= \llbracket P_2 \rrbracket_{PNK}
\end{aligned}
$$

Case $P_1 \vee P_2$.

$$\begin{aligned}
&\mathit{return}\ (\llbracket P_1 \vee P_2 \rrbracket^p\ ()\ A)\\
&= \mathit{return}\ (\llbracket P_1 \rrbracket^p\ ()\ A \cup \llbracket P_2 \rrbracket^p\ ()\ A)\\
&= \mathit{return}\ (\llbracket P_1 \rrbracket^p\ ()\ A) \gg\!= \lambda B \to \mathit{return}\ (\llbracket P_2 \rrbracket^p\ ()\ A) \gg\!= \lambda C \to \mathit{return}\ (B \cup C)\\
&= \llbracket P_1 \rrbracket_{PNK}\ A \gg\!= \lambda B \to \llbracket P_2 \rrbracket_{PNK}\ A \gg\!= \lambda C \to \mathit{return}\ (B \cup C)\\
&= \llbracket P_1 \vee P_2 \rrbracket_{PNK}\ A
\end{aligned}$$

□

LEMMA C.4. *Let C be a closed probabilistic computation. Define $\psi(m) = m \gg\!= \lambda\mu \to \mathit{return}\ \big(\mu(())\big)$. Then for all $A \in 2^{PH}$:*

$$\psi(\llbracket C \rrbracket\ ()\ A) = \llbracket C \rrbracket_{PNK}\ A$$

PROOF. By induction on the structure of probabilistic computations. Note that we have the following computation rules for ψ:

$$\begin{aligned}
\psi(\mathit{return}\ (\lambda() \to A)) &= \mathit{return}\ A\\
\psi(m_1 \gg\!= f) &= m_1 \gg\!= (\psi \circ f)\\
\psi(r \cdot m) &= r \cdot \psi(m)\\
\psi(m_1 + m_2) &= \psi(m_1) + \psi(m_2)\\
\psi(\bigsqcup_{i \geq 0} m_i) &= \bigsqcup_{i \geq 0} \psi(m_i)
\end{aligned}$$

Atomic Computations. For atomic computations C, we have

$$\begin{aligned}
\psi(\llbracket C \rrbracket\ ()\ A) &= \psi(\mathit{return}\ (\lambda() \to \{f_C\ ()\ h \mid h \in A\}))\\
&= \mathit{return}\ \{f_C\ ()\ h \mid h \in A\}\\
&\quad \text{Immediate for } f \leftarrow n \text{ and } \mathit{dup}\text{, for predicates this follows from Lemma ??}\\
&= \llbracket C \rrbracket_{PNK}\ A
\end{aligned}$$

Sequential composition.

$$\begin{aligned}
&\psi\,(\llbracket C_1 \,; C_2 \rrbracket\ ()\ A)\\
&=\psi\left(\llbracket C_1 \rrbracket\ ()\ A \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket\ ()\ \big(\bigcup_{x \in dom(\mu)} \mu(x)\big)\right)\\
&=\psi\,\big(\llbracket C_1 \rrbracket\ ()\ A \gg\!= \lambda\mu \to \llbracket C_2 \rrbracket\ ()\ \mu(())\big)\\
&=\psi\,\big(\llbracket C_1 \rrbracket\ ()\ A \gg\!= \lambda\mu \to \mathit{return}\ \mu(()) \gg\!= \llbracket C_2 \rrbracket\ ()\ \big)\\
&=\psi\,(\psi(\llbracket C_1 \rrbracket\ ()\ A) \gg\!= \llbracket C_2 \rrbracket\ ()\ )\\
&=\psi(\llbracket C_1 \rrbracket\ ()\ A) \gg\!= (\psi \circ \llbracket C_2 \rrbracket\ ()\ )\\
&=\llbracket C_1 \rrbracket_{PNK}\ A \gg\!= \llbracket C_2 \rrbracket_{PNK}
\end{aligned}$$

Parallel composition.

$$
\begin{aligned}
&\psi\,(\llbracket C_1 \,\&\, C_2 \rrbracket\ ()\ A)\\
=&\psi\left(\llbracket C_1 \rrbracket\ ()\ A \ggg= \lambda\mu_1 \to \llbracket C_2 \rrbracket\ ()\ A \ggg= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2)\right)\\
=&\psi\,(\llbracket C_1 \rrbracket\ ()\ A \ggg= \lambda\mu_1 \to \llbracket C_2 \rrbracket\ ()\ A \ggg= \lambda\mu_2 \to \mathit{return}\ (\lambda() \to \mu_1(()) \cup \mu_2(())))\\
=&\llbracket C_1 \rrbracket\ ()\ A \ggg= \lambda\mu_1 \to \llbracket C_2 \rrbracket\ ()\ A \ggg= \lambda\mu_2 \to \mathit{return}\ (\mu_1(()) \cup \mu_2(()))\\
=&\llbracket C_1 \rrbracket\ ()\ A \ggg= \lambda\mu_1 \to \mathit{return}\ \mu_1(()) \ggg= \lambda A_1 \to\\
&\llbracket C_2 \rrbracket\ ()\ A \ggg= \lambda\mu_2 \to \mathit{return}\ \mu_2(()) \ggg= \lambda A_2 \to \mathit{return}\ (A_1 \cup A_2)\\
=&\psi\,(\llbracket C_1 \rrbracket\ ()\ A) \ggg= \lambda A_1 \to \psi\,(\llbracket C_2 \rrbracket\ ()\ A) \ggg= \lambda A_2 \to \mathit{return}\ (A_1 \cup A_2)\\
=&\llbracket C_1 \rrbracket_{PNK}\ A \ggg= \lambda A_1 \to \llbracket C_2 \rrbracket_{PNK}\ A \ggg= \lambda A_2 \to \mathit{return}\ (A_1 \cup A_2)\\
=&\llbracket C_1 \,\&\, C_2 \rrbracket_{PNK}\ ()A
\end{aligned}
$$

Probabilistic Choice.

$$
\begin{aligned}
&\psi(\llbracket C_1 \oplus_r C_2 \rrbracket\ ()\ A)\\
=&\psi\,(r\llbracket C_1 \rrbracket\ ()\ A + (1-r)(\llbracket C_2 \rrbracket\ ()\ A))\\
=&r \cdot \psi\,(\llbracket C_1 \rrbracket\ ()\ A) + (1-r) \cdot \psi\,(\llbracket C_2 \rrbracket\ ()\ A)\\
=&r(\llbracket C_1 \rrbracket_{PNK}\ A) + (1-r)(\llbracket C_2 \rrbracket_{PNK}\ A)\\
=&\llbracket C_1 \oplus_r C_2 \rrbracket_{PNK}\ A)
\end{aligned}
$$

Iteration.

$$
\begin{aligned}
&\psi(\llbracket C^* \rrbracket\ ()\ A)\\
=&\psi(\bigsqcup_{n\geq 0}(\llbracket C^n \rrbracket\ ()\ A)\\
=&\bigsqcup_{n\geq 0} \psi(\llbracket C^n \rrbracket\ ()\ A)\\
=&\bigsqcup_{n\geq 0} \llbracket C^n \rrbracket_{PNK}\ A\\
=&\llbracket C^* \rrbracket_{PNK}\ A
\end{aligned}
$$

□

C.5 Theorem 6.2

Proof. We prove the following slightly stronger statement: Let R_1, R_2 be terminals such that $\Vdash_c R_1 : T$, $\Vdash_c R_2 : \mathcal{P}\,\mathbf{1}$ and $R_1 \rightsquigarrow R_2$, then $\llbracket R_1 \rrbracket = \llbracket R_2 \rrbracket$ if $T = \mathcal{P}\,\mathbf{1}$, otherwise $\llbracket R_2 \rrbracket = \llbracket R_1 \,;\, \mathit{skip} \rrbracket$.

We proceed by induction on the elaboration relation. In the first three cases of $\rightsquigarrow$, the result is immediate.

In what follows, assume $R_{11} \rightsquigarrow R_{21}$ and $R_{12} \rightsquigarrow R_{22}$, $\Vdash_c R_{11} : T_1$, $\Vdash_c R_{11} : T_2$,

If $R_{11} \,;\, R_{12} \rightsquigarrow R_{21} \,;\, R_{22}$, then $\llbracket R_{21} \,;\, R_{22} \rrbracket = \llbracket R_{11} \,;\, R_{22} \rrbracket$, by induction independently of T_1. Then either $\llbracket R_{21} \,;\, R_{22} \rrbracket = \llbracket R_{11} \,;\, R_{12} \rrbracket$ or $\llbracket R_{21} \,;\, R_{22} \rrbracket = \llbracket R_{11} \,;\, R_{12} \,;\, \mathit{skip} \rrbracket$ according to T_2.

If $R_{11} \,\&\, R_{12} \rightsquigarrow R_{21} \,\&\, R_{22}$, then the required follows immediately by induction, inversion and the definition of the denotational semantics.

If $R_{11} \oplus R_{12} \rightsquigarrow R_{21} \oplus R_{22}$, then Note that $T_1 = T_2$, the required result then either follows immediately, by induction, or by induction and distributivity of ; over $\oplus$.

Assume $R_1 \rightsquigarrow R_2$. It is easy to show by induction on n that if $[\![R_1]\!] = [\![R_2]\!]$, then $[\![R_1^n]\!] = [\![R_2^n]\!]$ for all n. It then follows that $[\![{R_1}^*]\!] = [\![{R_2}^*]\!]$

The penultimate case follows by inversion and the definition of the denotational semantics. The final case is immediate after inversion. □

C.6 Theorem 6.4

Proof. The proof proceeds by induction on the structure of the evaluation rules and case analysis on the final rule.

In cases E-Pred, E-Mod, E-Dup, E-Prod and E-Abs, C is a terminal. By reflection, $C = R$, Hence $[\![C]\!] = [\![R]\!]$.

In cases E-Seq, E-Par, E-Choice, the required follows by unfolding the definition and applying the induction hypothesis.

Case E-Iter. Note that

$$[\![C^*]\!]\ \rho\ A = \bigsqcup_{n\geq 0} [\![C^n]\!]\ \rho\ A \text{ and} [\![R^*]\!]\ \rho\ A = \bigsqcup_{n\geq 0} [\![R^n]\!]\ \rho\ A$$

Then we we need to show that $\forall n \geq 0 : [\![C^n]\!] \;=\; [\![R^n]\!]$. This follows by induction on n.

Case E-Force.

$$[\![\mathit{force}\ \mathit{thunk}\ C]\!]\ \rho\ A = [\![\mathit{thunk}\ C]\!]\ \rho\ A = [\![C]\!]\ \rho\ A = [\![R]\!]\ \rho\ A$$

Case E-AppAbs.

$$
\begin{aligned}
& [\![C_1\ V]\!]\ \rho\ A \\
& \quad \text{by definition} \\
= & [\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
& \quad \text{by induction} \\
= & [\![\lambda x\!:\!S.C_{11}]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
& \quad \text{by definition} \\
= & \mathit{return}\ (\lambda(\lambda v \to [\![C_{11}]\!]\ [x \mapsto v]\rho\ ) \to A) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
& \quad \text{by definition} \\
= & \Xi\{(\lambda v \to [\![C_{11}]\!]\ [x \mapsto v]\rho\ )\ ([\![V]\!]\ \rho)\ A\} \\
& \quad \beta\text{-reduction} \\
= & [\![C_{11}]\!]\ [x \mapsto ([\![V]\!]\ \rho)]\rho\ A \\
& \quad \text{Lemma ??} \\
= & [\![[x \mapsto V]C_{11}]\!]\ \rho\ A \\
& \quad \text{by induction} \\
= & [\![R]\!]\ \rho\ A
\end{aligned}
$$

Case E-AppSeq.

$$
\begin{aligned}
&[\![C_1\ V]\!]\ \rho\ A \\
&\quad \text{by definition} \\
=&[\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{by induction} \\
=&[\![R_{11}\,;R_{12}]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{by definition} \\
=&[\![R_{11}]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![R_{12}]\!]\ \rho\ \Big(\bigcup_{x \in \mathit{dom}(\mu_1)} \mu_1(x)\Big) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{associativity of } \gg\!=\text{, by definition} \\
=&[\![R_{11}]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![R_{12}\ V]\!]\ \rho\ \Big(\bigcup_{x \in \mathit{dom}(\mu_1)} \mu_1(x)\Big) \\
&\quad \text{by induction} \\
=&[\![R_{11}]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![R_2]\!]\ \rho\ \Big(\bigcup_{x \in \mathit{dom}(\mu_1)} \mu_1(x)\Big) \\
&\quad \text{by definition} \\
=&[\![R_{11}\,;R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-AppChoice.

$$
\begin{aligned}
&[\![C_1\ V]\!]\ \rho\ A \\
&\quad \text{by definition} \\
=&[\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{by induction} \\
=&[\![R_{11} \oplus R_{12}]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
=&\big(r([\![R_{11}]\!]\ \rho\ A) + (1-r)([\![R_{12}]\!]\ \rho\ A)\big) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{distributivity} \\
=&r([\![R_{11}]\!]\ \rho\ A) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\ + (1-r)([\![R_{12}]\!]\ \rho\ A)\big) \gg\!= \lambda\mu \to \Xi\{f\ ([\![V]\!]\ \rho\ )\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&\quad \text{by definition} \\
=&r([\![R_{11}\ V]\!]\ \rho\ A) + (1-r)([\![R_{12}\ V]\!]\ \rho\ A) \\
&\quad \text{by induction} \\
=&r([\![R_1]\!]\ \rho\ A) + (1-r)([\![R_2]\!]\ V\ \rho A) \\
&\quad \text{by definition} \\
=&[\![R_1 \oplus R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToProduce.

$$
\begin{aligned}
&[\![C_1 \ \mathit{to}\ x.C_2]\!]\ \rho\ A \\
&\quad \text{by definition} \\
=&[\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [x \mapsto v]\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&\quad \text{by induction} \\
=&[\![\mathit{produce}\ V]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [x \mapsto v]\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&\quad \text{by definition} \\
=&\mathit{return}\ (\lambda[\![V]\!]\ \rho \rightarrow A) \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [x \mapsto v]\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&\quad \text{left-unit of } \gg\!= \\
=&\Xi\{[\![C_2]\!]\ [x \mapsto ([\![V]\!]\ \rho)]\rho\ A\} \\
=&[\![C_2]\!]\ [x \mapsto ([\![V]\!]\ \rho)]\rho\ A \\
&\quad \text{by Lemma ??} \\
=&[\![[x \mapsto V]C_2]\!]\ \rho\ A \\
&\quad \text{by induction} \\
=&[\![R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToSeq. like E-AppSeq

Case E-ToChoice. like E-AppChoice

Case E-ToIter.

$$
\begin{aligned}
&[\![C_1 \ \mathit{to}\ x.C_2]\!]\ \rho\ A \\
=&[\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
=&[\![{R_1}^*]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&\quad \text{by inversion} \\
=&[\![{R_1}^*]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow [\![C_2]\!]\ [x \mapsto ()]\rho\ \mu(()) \\
&\quad \text{Lemma ??} \\
=&[\![{R_1}^*]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow [\![[x \mapsto \mathit{unit}]C_2]\!]\ \rho\ \mu(()) \\
=&[\![{R_1}^*]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow [\![[x \mapsto \mathit{unit}]C_2]\!]\ \rho\ \Big(\bigcup_{x \in \mathit{dom}(\mu)} \mu(x)\Big) \\
&\quad \text{by induction} \\
=&[\![{R_1}^*]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow [\![R_2]\!]\ \rho\ \Big(\bigcup_{x \in \mathit{dom}(\mu)} \mu(x)\Big) \\
&\quad \text{by definition} \\
=&[\![{R_1}^*\, ; R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToPar. Note that the inversion for $\Gamma \Vdash_c C_1 \,\&\, C_2 : T$ says that $T = \mathcal{P}\,\mathbf{1}$. Then the argument proceeds analogously to E-ToIter. ∎

Lemma C.5. *Let $V, \widehat{V}$ be a values and C be a computation such that $x : \widehat{S}, \Gamma \vdash_v V : S$, $\Gamma \vdash_v \widehat{V} : \widehat{S}$ and $x : \widehat{S}, \Gamma \vdash_c C : T$. Let $\rho[\![\Gamma]\!]$ then*

$$[\![V]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho = [\![[x \mapsto \widehat{V}]V]\!]\ \rho$$
$$[\![C]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho = [\![[x \mapsto \widehat{V}]C]\!]\ \rho$$

Proof. By induction on the structure of the typing derivation. We can ignore T-Unit, T-Header, T-Lit, T-Skip,T-Drop,T-Dup, since they don't contain any variables.

Case T-Thunk $\qquad V = \mathit{thunk}\ C \qquad \dfrac{\Gamma \vdash_c C : T}{\Gamma \vdash_v \mathit{thunk}\ C : \mathcal{T}\,T}$

$$[\![\mathit{thunk}\ C]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\ = [\![C]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho] = [\![[x \mapsto \widehat{V}]C]\!]\ \rho = [\![[x \mapsto \widehat{V}]\mathit{thunk}\ C]\!]\ \rho$$

Case Atomic Computations $\qquad C = P$ or $V_1 \leftarrow V_2$ or $\mathit{dup} \qquad \dfrac{\vdots}{\Gamma \vdash_c C : \mathcal{P}\,\mathbf{1}}$

For predicates we can prove:

- Negation

$$B_{\neg P}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho = \neg B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho = \neg B_{[x \mapsto \widehat{V}]P}\ \rho = B_{\neg [x \mapsto \widehat{V}]P}\ \rho = B_{[x \mapsto \widehat{V}]\neg P}\ \rho$$

- Disjunction

$$\begin{aligned} & B_{P_1 \vee P_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \\ =& B_{P_1}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \text{ or } B_{P_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \\ =& B_{[x \mapsto \widehat{V}]P_1}\ \rho \text{ or } B_{[x \mapsto \widehat{V}]P_2}\ \rho \\ =& B_{[x \mapsto \widehat{V}(P_1 \vee P_2)}\ \rho \end{aligned}$$

  And similar for T-Conj

- Guard

$$\begin{aligned} & B_{V_1 = V_2}\ ([x \mapsto [\![\widehat{V}]\!]\ ]\rho)\ (\pi :: h) \\ =& \pi.[\![V_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho = [\![V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho \\ =& \pi.[\![[x \mapsto \widehat{V}]V_1]\!]\ \rho[\![[x \mapsto \widehat{V}]V_2]\!]\ \rho \\ =& B_{[x \mapsto \widehat{V}]V_1 = V_2}\ \rho\ (pi :: h) \end{aligned}$$

Hence, $B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho = B_{[x \mapsto \widehat{V}]P}\ \rho$, and

$$\begin{aligned} f_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ h &= \begin{cases} h & \text{if} B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ h \\ \bot & \text{otherwise} \end{cases} \\ &= \begin{cases} h & \text{if} B_{[x \mapsto \widehat{V}]P}\ \rho\ h \\ \bot & \text{otherwise} \end{cases} \\ &= f_{[x \mapsto \widehat{V}]P}\ \rho\ h \end{aligned}$$

For T-Mod:

$$\begin{aligned} f_{V_1 \leftarrow V_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ (\pi :: h) &= \pi[[\![V_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \mapsto [\![V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho] :: h \\ &= \pi[[\![[x \mapsto \widehat{V}]V_1]\!]\ \rho \mapsto [\![[x \mapsto \widehat{V}]V_2]\!]\ \rho] :: h \\ &= f_{[x \mapsto \widehat{V}]V_1 \leftarrow V_2}\ \rho\ (\pi :: h) \end{aligned}$$

Thus we can conclude that for an atomic computation C,

$$\begin{aligned}\llbracket C\rrbracket\ [x\mapsto \llbracket\widehat{V}\rrbracket\ \rho]\rho\ A &= return\ (\lambda()\to\{f_C\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ h\mid h\in A\})\\ &= return\ (\lambda()\to\{f_{[x\mapsto\widehat{V}]C}\ \rho\ h\mid h\in A\})\\ &= \llbracket[x\mapsto\widehat{V}C]\rrbracket\ \rho\ A\end{aligned}$$

Case T-SEQ $\qquad C = C_1\,;C_2 \qquad \dfrac{\Gamma\vdash_c C_1 : T_1 \qquad \Gamma\vdash_c C_2 : T_2}{\Gamma\vdash_c C : T_2}$

$$\begin{aligned}\llbracket C_1\,;C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A &= \llbracket C_1\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A \gg\!= \lambda\mu\to\llbracket C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ \bigcup_{y\in dom(\mu)}\mu(y)\\ &\text{by induction}\\ &= \llbracket[x\mapsto\widehat{V}]C_1\rrbracket\ \rho\ A\gg\!=\lambda\mu\to\llbracket[x\mapsto\widehat{V}]C_2\rrbracket\ \rho\ \bigcup_{y\in dom(\mu)}\mu(y)\\ &= \llbracket[x\mapsto\llbracket\widehat{V}\rrbracket\ ](C_1\,;C_2)\rrbracket\ \rho\ A\end{aligned}$$

Case T-PAR $\qquad C = C_1\,\&\,C_2 \qquad \dfrac{\Gamma\vdash_c C_1 : T \qquad \Gamma\vdash_c C_2 : T}{\Gamma\vdash_c C : T}$

$$\begin{aligned}&\llbracket C_1\,;C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A\\ &= \llbracket C_1\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A\gg\!=\lambda\mu_1\to\llbracket C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A\gg\!=\lambda\mu_2\to return\ (\mu_1\bigvee\mu_2)\\ &\text{by induction}\\ &= \llbracket[x\mapsto\widehat{V}]C_1\rrbracket\ \rho\ A\gg\!=\lambda\mu_1\to\llbracket[x\mapsto\widehat{V}]C_2\rrbracket\ \rho\ A\gg\!=\lambda\mu_2\to return\ (\mu_1\bigvee\mu_2)\\ &= \llbracket(C_1\,\&\,C_2)[x\mapsto\widehat{V}]\rrbracket\ \rho\ A\end{aligned}$$

Case T-CHOICE $\qquad C = C_1\oplus C_2 \qquad \dfrac{\Gamma\vdash_c C_1 : T \qquad \Gamma\vdash_c C_2 : T}{\Gamma\vdash_c C : T}$

$$\begin{aligned}&\llbracket C_1\oplus C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A\\ &= r\llbracket C_1\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A + (1-r)(\llbracket C_2\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A)\\ &= r\llbracket[x\mapsto\widehat{V}]C_1\rrbracket\ \rho\ A + (1-r)(\llbracket[x\mapsto\widehat{V}]C_2\rrbracket\ \rho\ A)\end{aligned}$$

Case T-PRODUCE $\qquad C = produce\ V \qquad \dfrac{\Gamma\vdash_v V : S}{\Gamma\vdash_c produce\ V : T}$

$$\begin{aligned}&\llbracket produce\ V\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\ A\\ &= return\ (\lambda\llbracket V\rrbracket\ [x\mapsto\llbracket\widehat{V}\rrbracket\ \rho]\rho\to A) \qquad = return\ (\lambda\llbracket[x\mapsto\widehat{V}]V\rrbracket\ \rho A\to)\\ &= \llbracket[x\mapsto\widehat{V}]produce\ V\rrbracket\ \rho\ A\end{aligned}$$

Case T-Force $\quad C = \mathit{force}\ V \quad \dfrac{\Gamma \vdash_v V : \mathcal{T}\, T}{\Gamma \vdash_c \mathit{force}\ V : T}$

$$\begin{aligned}
&[\![\mathit{force}\ V]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
&= [\![V]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
&= [\![[x \mapsto \widehat{V}]V]\!]\ \rho\ A \\
&= \llbracket [x \mapsto \widehat{V}](\mathit{force}\ V) \rrbracket\ \rho\ A
\end{aligned}$$

Case T-Abs $\quad C = \lambda x : S.C' \quad \dfrac{x : S, \Gamma \vdash_c C' : T}{\Gamma \vdash_c \lambda X : S.C' : S \to T}$

$$\begin{aligned}
&[\![\lambda y : S.C']\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
&= \mathit{return}\ (\lambda\lambda v \to [\![C']\!]\ [y \mapsto v, x \mapsto [\![\widehat{V}]\!]\ \rho]\rho A \to) \\
&= \mathit{return}\ (\lambda\lambda v \to [\![[x \mapsto \widehat{V}]C']\!]\ [y \mapsto v]\rho A \to) \\
&= \llbracket [x \mapsto \widehat{V}](\lambda y : S.C') \rrbracket\ \rho\ A
\end{aligned}$$

Case T-Iter $\quad C = {C_1}^* \quad \dfrac{\Gamma \vdash_c C_1 : \mathcal{P}\,\mathbf{1}}{\Gamma \vdash_c {C_1}^* \mathcal{P}\,\mathbf{1} :}$

$$\begin{aligned}
&[\![{C_1}^*]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho\ A \\
&= \bigsqcup_{n \geq 0} [\![C_1^n]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho\ A \\
&= \bigsqcup_{n \geq 0} [\![[x \mapsto \widehat{V}]C_1^n]\!]\ \rho\ A \\
&= \llbracket [x \mapsto \widehat{V}]({C_1}^*) \rrbracket\ \rho\ A
\end{aligned}$$

Case T-App $\quad C = C'\ V \quad \dfrac{\Gamma \vdash_c C : S \to T \qquad \Gamma \vdash_v V : S}{\Gamma \vdash_c C'\ V : T}$

$$\begin{aligned}
&[\![C'\ V]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \\
&= [\![C']\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \gg\!= \lambda\mu \to \Xi\{f\ [\![V]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&= [\![[x \mapsto \widehat{V}]C']\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{f\ [\![[x \mapsto \widehat{V}]V]\!]\ \rho\ \mu(f) \mid f \in \mathit{dom}(\mu)\} \\
&= [\![[x \mapsto \widehat{V}]C'\ V]\!]\ \rho\ A
\end{aligned}$$

Case T-To $\quad C = C_1 \ \mathit{to}\ y.C_2 \quad \dfrac{\Gamma \vdash_c C_1 : \mathcal{P}\, S \qquad y : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \ \mathit{to}\ y.C_2 : T}$

$$
\begin{aligned}
&[\![C_1 \ \mathit{to}\ y.C_2]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \\
&= [\![C_1]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \gg\!= \lambda\mu \to \Xi\{[\![C_2]\!]\ [y \mapsto v, x \mapsto [\![V]\!]\ \rho]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&= [\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{[\![[x \mapsto \widehat{V}]C_2]\!]\ [y \mapsto v]\rho\ \mu(v) \mid v \in \mathit{dom}(\mu)\} \\
&= [\![[x \mapsto \widehat{V}](C_1 \ \mathit{to}\ y.C_2)]\!]\ \rho\ A
\end{aligned}
$$

□

Since $\Gamma \Vdash_v V : S$ implies $\Gamma \vdash_v V : S$ and $\Gamma \Vdash_c C : T$ implies $\Gamma \vdash_c C : T$, it follows that:

Lemma C.6. *Let $V, \widehat{V}$ be a values and C be a computation such that $x : \widehat{S}, \Gamma \Vdash_v V : S$, $\Gamma \Vdash_v \widehat{V} : \widehat{S}$ and $x : \widehat{S}, \Gamma \Vdash_c C : T$. Let $\rho[\![\Gamma]\!]$ then*

$$
\begin{aligned}
[\![V]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho &= [\![[x \mapsto \widehat{V}]V]\!]\ \rho \\
[\![C]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho &= [\![[x \mapsto \widehat{V}]C]\!]\ \rho
\end{aligned}
$$