Universiteit Leiden

Master Computer Science

Operational Semantics of GHOPFL

A guarded, higher order, probabilisitc, coinductive, functional language

Name:	Rintse van de Vlasakker
Student ID:	1903748
Date:	05/07/2021
Specialisation:	Foundations of Computing
1st supervisor:	Dr. Henning Basold
2nd supervisor:	Dr. Alfons Laarman

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

We introduce GHOPFL: a functional, higher order, probabilistic and coinductive language with guarded recursion. The guarded recursion ensures a desirable property for coinductive programs (i.e. programs with infinite data): productivity. A productive program, even if it produces infinitely many results, produces them a finite amount of time apart. Productivity alleviates many problems commonly associated with infinite data. We show how probabilistic programming benefits from coinduction and producitivity in multiple ways. We give both a small-step and a big-step operational semantics for GHOPFL and prove that they are equivalent. This means that one may choose the theoretically useful small-step semantics or the implementation-related big-step semantics as a starting point when reasoning about the language. Finally, we give example programs that make use of the features of the language. These examples may be run and examined in the provided interpreter, implemented in Haskell.

Contents

1 Introduction

Probabilistic programming is an increasingly common tool for statistical modelling. It takes concepts from programming and applies them to statistical model building. Its many applications, ranging from computer vision [17] to inductive program synthesis [20], have proven its usefulness as a programming regime. One of the basic concepts of probabilistic programming is distribution sampling. Using simple distribution sampling primitives, a programmer may build up more complex models and perform analysis on them. For example, using only samples from a uniform distribution over $[0, 1]$ it is possible to generate a Poisson-distributed variate with mean λ using the following procedure, commonly attributed to Knuth [14]:

```
procedure POISSON(λ)
    k ← 0,    p ← 1
    repeat
        k ← k + 1
        p ← p × (sample [0, 1])
    until p ≤ e^−λ
    return  k − 1
end procedure
```

The procedure rests on the fact that the number of independent exponential random variables (with mean 1) to be added before exceeding some fixed number λ is Poisson distributed (with mean λ). There exists a way to transform a uniform random variable into an exponential variable: $(-\ln U)$ is exponential (with mean 1) when U is uniform in $[0, 1]$ (see [12], Lemma 3.22). Using this insight, we might also multiply random uniform samples from $[0, 1]$ until we have reached $e^{-\lambda}$ to obtain a Poisson distribution. It is clear to see that this procedure depends on a looping or recursion construct, illustrating the utility of programming constructs for statistical modelling purposes.

Another useful feature for a programming language is the support for infinite data. Many natural objects and processes are best modelled as infinite data and being able to work on such infinite data can be a powerful tool. Going back to the Poisson example from before: One might imagine wanting to know the mode (i.e. the most frequent value) of our newly defined distribution. One way to estimate it, is to sample more and more values while keeping track of the frequencies of the samples so far. The more values we sample, the more certain we can be that the mode of our samples is the mode of the distribution. As there is no point at which can be totally certain, we are dealing with a coinductive (i.e. infinite) process. Instead of having to calculate the mode based on some fixed amount of samples, coinductive languages support the definition of this estimation process as a whole (from which finite parts might still be extracted).

Now say we want to plot 10 histograms from this mode estimation process, plotting every 100 steps. If we had hard-coded some finite list of estimation steps, we would have to make sure that the list is at least 1000 steps long. Furthermore, any steps beyond 1000 are not used for the plots, causing inefficiencies if calculated. As a solution, we might parametrise the estimation list generation by how many elements we expect we will need. Now the programmer has to work out the relations between the various processes in their program manually. It is easy to see that a plotting function like this would need 100 steps from the optimisation process to produce one result and will therefore use $100 \times 10 = 1000$ elements in total. However, as the complexity of the processes increases, working

out these relations becomes a bothersome task. In the coinductive regime, we can simply define a function that calculates the values it needs from the infinite optimisation process, never running out of data and never evaluating too much of it. This reduces the complexity (and possibly improves the efficiency) of such a plotting program. In general, some processes are naturally infinite and keeping this property when trying to represent them programmatically makes programming more intuitive. Furthermore, reasoning about programs in a theoretical setting becomes easier, as the description of an infinite process is typically much smaller than some finite approximation thereof.

When programming with infinite processes, there is a new set of problems that programmers have to be cautious of. It is, for example, easy to write programs that are ill-defined, like folds on infinite data: What is the sum of an infinite list of integers? But there are are more subtle considerations to be made as well. Crucial for this thesis is program productivity. A program is productive if, even when producing infinite data, all parts of this data become available within finite time of each other. In coinductive languages, productivity may be seen as a close relative to termination guarantees. Productivity is desirable, as it alleviates many problems to do with combining calculations.

We have already encountered possible non-productivity in the Poisson generator procedure: If we get an infinite amount of ones from our random distribution, the procedure never terminates, as p never gets closer to $e^{-\lambda}$. As a result, our mode estimation process using this procedure also gets stuck. It is possible to write a similar program that will always give results, by using *guards* when recursing or looping. The idea behind such *guards* is that they can not be evaluated over and may thus stop evaluation of programs that would otherwise go on forever.

This thesis defines operational semantics for a guarded, higher order, productive, probabilistic programming language that supports distribution sampling, as well as some forms of coinductive data. By enforcing productivity at the level of types, we avoid some of the problems such data normally brings. This work was prompted by the denotational semantics for a similar language by Birkedal et al. [5], Henning Basold [4] and more. The approach taken was mainly inspired by Borgström et al. [6], Clouston et al. [8] and Abel and Vezzosi [1]. We give both a small-step semantics that is easy to reason about theoretically and a big-step semantics that corresponds more closely with the execution of programs. We proof that these two semantics are equivalent, meaning that one may choose either as a starting point if they want to reason about properties of the language. Finally we give some example programs and elaborate on their execution. We provide an interpreter [24] based on the big step semantics, in which the example programs may be examined.

We start by providing some background knowledge in Section 2. We give the syntax for the language in Section 3. We then define a small-step semantics and a big-step semantics in Sections 4 and 5 respectively. In Section 6, we proof the equivalence of these two semantics. We discuss related work in Section 7. Finally, example programs are given and explained in detail in Section 8.

2 Background

We introduce some background topics that will aid in describing the semantics for our proposed language. We start by explaining some of the major concepts at the foundations of the thesis like semantics itself and the lambda calculus, but we also give some more technical definitions of concepts relating to the details of the semantics.

2.1 Semantics

There are two main approaches to defining the semantics of a programming language: Denotational semantics and operational semantics. The former expresses the meaning of a program in terms of mathematical objects representing expressions and manipulations thereof. The latter expresses the meaning of a program through logical statements about its execution. Denotational semantics tend to be more powerful, as properties of the mathematical objects used can aid in proving properties of the programs, but are harder to write down. Operational semantics tend to be easier to write down and are more closely related to the execution of programs. This means they are also more useful as a basis for implementations of a language. We will be concerning ourselves with operational semantics in this thesis.

Big-step and small-step Within operational semantics, there are two ways of constructing proofs: big-step and small-step. Big-step semantics (a.k.a. natural semantics) describe how a calculation can be broken down into smaller sub-calculations and how the results of these should be combined. Small-step semantics only provide rules specifying how all the parts in a program should evolve. The meaning of a program is then constructed through repeated application of the appropriate rules. We will give both the small-step and big-step semantics for our proposed language in this thesis.

Terms and types We will be using a typing system to give our semantics. Under such a system, we usually speak of two related concepts: the *term* and its *type*. A *term* is some piece of a program, that itself may be viewed as a program. Terms are closely related to syntax. In the following generic program, 1 is a term.

$$\texttt{if } \textbf{true} \texttt{ then } 1 \texttt{ else } 0.$$

So is (`if` **true** `then` 1 `else` 0). However, (`if` `true`) is not, as it would not form a valid program on its own. The typing system assigns a *type* to every term. In this example, the term (1) is of type **int**, the term (**true**) is of type **bool** and the entire program is of type **int** again.

2.2 Lists

As we will be using lists in our operational semantics, as well as in example programs, we now define some notation and concepts regarding lists. We denote lists with $[1, 2, \ldots]$. We define some common operations on lists:

- Head takes the first element: $\texttt{head}\ [1, 2, 3] = 1$.
- Tail takes everything but the first element: $\texttt{tail}\ [1, 2, 3] = [2, 3]$.
- Null returns whether a list is empty: $\texttt{null}\ [\,] = \texttt{true}$, $\texttt{null}\ [1, 2, 3] = \texttt{false}$.

- The operator : constructs a list from a head and a tail: $1 : [2,3] = [1,2,3]$ (also referred to as `cons` in prefix notation).
- The operator $+\!\!+$ appends two lists: $[1] +\!\!+ [2,3] = [1,2,3]$ (also referred to as `append` in prefix notation).
- The function (`map` f) applies the function f to each element in a list: `map` $(+1)$ $[1,2,3] = [2,3,4]$. Here, $(+1)$ denotes the successor function on integers.
- The function `foldl` f v folds a list into a single value by applying f to an intermediate result (initially v) and the next element in the list: `foldl` $(+)$ 0 $[1,2,3] = ((0+1)+2)+3 = 6$. Here, $(+)$ denotes a prefix notation function for addition on integers, like in Haskell syntax.

2.3 Products and coproducts

We will apply the notions of products and coproducts only on types. We will use sets to represent types in the following manner: All terms of a given type are considered as a set with the name of the type. For example, the boolean type is represented by a set $\mathbf{bool} = \{\mathsf{true}, \mathsf{false}\}$. Because of this, our explanation will focus on set theory, but it should be noted that these notions can be generalized in category theory.

Products and coproducts are dual notions that reflect two ways in which two types may be combined. Many programmers will be familiar with products, as many languages support pairs of some sort. In Haskell pairs are created using $(,)$, in C++ there is `std::pair`, etc. We will be using the Haskell notation. Let us consider the pairing of an integer and a real number: $(1, 1.0)$. We say that we *insert* 1 and 1.0 into the product $(1, 1.0)$. This new term has the product type $(\texttt{Int} \times \texttt{Double})$, where `Int` and `Double` are sets containing all integers and doubles respectively and $\times$ denotes the Cartesian product. One might think of this new type intuitively as containing elements that are *both* an integer *and* a double. There exist operations (called `fst` and `snd` in Haskell) that *project* one of the two values from a product, giving a value of type `Int` or `Double` again.

As mentioned before, coproducts are the dual notion to products. They may be though of as the disjoint union of two types, usually denoted $A + B$. Haskell supports coproducts through its constructors, which allow values of multiple types. An example is shown below.

$$\mathsf{data\ Number} = \mathsf{Integer\ Int} \mid \mathsf{Real\ Double}$$

Given this data definition, both the terms (Integer 1) and (Real 1.0) create a "Number" of type $(\texttt{Int} + \texttt{Double})$. We say that these terms *inject* values into the coproduct. One might think of this new type intuitively as containing elements that are *either* an integer *or* a real number. Haskell's `case` term allows us to pattern match a term of a coproduct type and *extract* a term of type `Int` or `Double` again.

2.4 Probabilistic programming

Probabilistic programming is a programming paradigm in which statistical modelling is the central focus. A programmer may build models, or manipulate built-in models, after which inference can be done (partially) automatically on said models. Examples of probabilistic programming

languages are *Anglican* [23] *Problog* [16], *Church* [9] and *Stan* [22]. The introduction already gave a typical model that a programmer may build in a probabilistic language. In the case of this Poisson distribution, most properties are understood quite well mathematically and many results we might find programmatically can also quite easily be calculated mathematically. The power of probabilistic programming is that much more complex models, combining many smaller models and processes using programming constructs, can be built and reasoned about just as easily as our Poisson example without the need for mathematical expertise and copious amounts of manual labour. The code base [24] contains such an example, combining multiple distributions in various ways, to construct a real-world scenario. In section 8, we will define and perform analysis on another standard example of a probabilistic programming model: the geometric distribution.

2.5 (Untyped) Lambda calculus

Lambda calculus is a computational system with very simple rules, introduced by Alonzo Church in the 1930s. Despite its simplicity, it is Turing-complete, making it a great foundation for many kinds of computing systems. It consists of three main concepts: variables, abstraction and application. Variables are denoted x, y, z. Abstraction and application together form the main computation mechanic of the lambda calculus. An abstracted term of the form $(\lambda x.\ M)$ can be used as the left operand of an application, which is simply denoted by putting terms next to each other: $((\lambda x.\ t)\ y)$. More formally, the grammar of lambda calculus is:

$$
\begin{array}{lll}
t, s ::= x & & \text{(Variables)} \\
\quad \mid\ \lambda x.\ t & & \text{(Abstraction)} \\
\quad \mid\ t\ \ s & & \text{(Application)}
\end{array}
$$

Grammar 1: Terms in lambda calculus.

In an abstraction like $(\lambda x.\ t)$, we call all occurrences of x within t *bound* by the binder $(\lambda x.)$. A variable is *free*, if it is not bound. Furthermore, a term is *closed* if it contains no free variables and *open* if it does. Terms are evaluated according to two major concepts: α-renaming and β-reduction. When α-renaming a term, we simply rename a bound variable. In the term $\lambda x.\ (x\ \ y)$, α-renaming x to z would yield $\lambda z.\ (z\ \ y)$. Care must be taken not to confuse variables of the same name, that in fact do not refer to the same variable. The term $(\lambda x.\ \lambda x.\ x)$, for example, may be α-renamed to $(\lambda y.\ \lambda x.\ x)$, but not to $(\lambda y.\ \lambda x.\ y)$, as the right-most x in the original term is bound to the closest binder. The actual function application, called β-reduction, may occur when we have a term of the form $((\lambda x.\ t)\ y)$. Reduction is performed by removing the binder $(\lambda x.)$ and substituting y for x in t. We use the notation $t[y/x]$ to mean the *capture-avoiding substitution* of y for all free x in t.

$$(\lambda x.\ t)\ y \to t[y/x]. \quad (\beta\text{-reduction})$$

Capture avoiding means that the substitution may not cause free variables to get bound by existing binders. Once again, care must be taken not to confuse binders. For example, while $((\lambda x.\ \lambda y.\ x)\ \ z)$ reduces to $(\lambda y.\ z)$, $((\lambda x.\ \lambda x.\ x)\ \ z)$ simply reduces to $(\lambda x.\ x)$.

Reduction strategy When defining a semantics for a lambda calculus, one has to decide what to do with function arguments when evaluating a function. There are two main strategies: *call-by-value* and *call-by-name*. Under the call-by-value strategy, function arguments must first be reduced to so

called *values* before they are passed to the function. Under the call-by-name strategy, the function arguments are simply substituted into the function body. If we define our values to only contain variables, the lambda term $(\lambda x.\ x)\ ((\lambda y.\ y)\ z)$ evaluates differently, depending on the evaluation strategy chosen:

$$(\lambda x.\ x)\ ((\lambda y.\ y)\ z) \to (\lambda x.\ x)\ (z) \to z \qquad \text{(Call-by-value)}$$
$$(\lambda x.\ x)\ ((\lambda y.\ y)\ z) \to (\lambda y.\ y)\ z \to z \qquad \text{(Call-by-name)}$$

The argument for the function $\lambda x.\ x$, is itself an application. Under a call-by-value strategy, as we only allow variables as values, we first have to reduce $((\lambda y.\ y)\ z)$ to a variable, before we may apply the function $\lambda x.\ x$. Under the call-by-name strategy, we simply apply β-reduction, substituting $((\lambda y.\ y)\ z)$ for x in its entirety. Note that the end result the same regardless of strategy. In general, any time two evaluation strategies lead to an answer, by the Church-Rosser Theorem [7], they lead to the same answer. It should be noted that termination of evaluation strategies is not guaranteed, however. A reduction system, is called *confluent* if the order of evaluation does not matter for the end result.

At this point, we have explained the general concepts behind the lambda calculus as a computational system. A more comprehensive tutorial of the lambda calculus was given by Raul Rojas [21].

Higher order languages A higher order language is one in which objects of function types can be passed around and used like any other object. Lambda calculus is such a higher order language. For the purposes of illustration, we will assume a lambda calculus with arithmetic operations and list types (not present in the examples so far). Within such a lambda calculus, we may write higher order functions, such as `map` (see Section 2.2). The list function `map` takes a function of type $A \to B$ and a list of type A, and applies the function to each element in the list, giving a list of type $[B]$. Assume we have a function that converts inches to centimetres: $\texttt{convert} : \mathbf{Inch} \to \mathbf{Cm}$ and we are given a list of measurements in inches: $\texttt{m} \stackrel{\text{def}}{=} [1.0\,\text{in}, 2.0\,\text{in}, 3.0\,\text{in}]$. We can easily convert the entire list into $[2.54\,\text{cm}, 5.08\,\text{cm}, 7.62\,\text{cm}]$, using (`map convert m`). One might think of its implementation as:

$$\texttt{map}\ f\ l \stackrel{\text{def}}{=} \textsf{if}\ (\texttt{null}\ l)\ \textsf{then}\ l\ \textsf{else}\ (f\ (\texttt{head}\ l))\ :\ (\texttt{map}\ f\ (\texttt{tail}\ l))$$

It is clear to see that the variable f is used as a function, yet it is passed around (in the recursive call, for example) like any other variable. Our language will be higher order and will thus support functions like `map`.

2.6 Simply typed lambda calculus

Simply typed lambda calculus (also introduced by Church) is a more restrictive calculus that alleviates some of the problems of an untyped lambda calculus, at the cost of having less computational power. Whereas untyped lambda calculus is Turing complete, simply typed lambda calculus is not. This decrease in power is caused by the fact that types may not recurse, as the only allowed type-formers are the function type-former $\to$, the product type-former $\times$ and the coproduct type former $+$ (see Section 2.3). Because of this, it is a useful starting point for computational systems, as it has many desirable properties. The most important of these properties for this work, is the fact that termination of evaluation is guaranteed on all well-typed terms, which is not the case for the untyped lambda calculus. In a simply typed lambda calculus, many of the rules from lambda

calculus still apply. We may still use α-renaming and β-reduction to evaluate terms, but the terms that we allow are narrowed down by *term formation rules*. Furthermore, additional evaluation rules are introduced for the new kinds of terms.

Term formation rules Term formation rules describe how terms may be combined into larger terms. An example of a type formation rule for a simply typed lambda calculus is given below:

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B}$$

Here, Γ is a typing context: a set of tuples containing a variable x and its type A, denoted $(x : A)$. Intuitively, this rule for function application can be read as: Given a term t that is of a function type $A \to B$ and a term s that is of the domain type of t, we may apply the function t to s to obtain a value in the codomain type of t. A program is well-typed, if all terms combine in a way that agrees with the term formation rules of the language.

Type formation rules To define how smaller types may be combined into larger types using the available type formers, we can write *type formation rules*. The type formation rule for the function type former $(\to)$ is given below:

$$\frac{\Delta \Vdash A : \mathbf{Ty} \quad \Delta \Vdash B : \mathbf{Ty}}{\Delta \Vdash A \to B : \mathbf{Ty}}$$

Here, Δ is a typing context: a set of type variables. Intuitively, this rule may be read as: Given that the language supports terms of type A and of type B, the language also supports terms of type $A \to B$.

2.7 Coinductive programming

Coinductive programming or co-programming, is a programming regime in which the programmer can create and manipulate infinite data. Coinduction is the mathematical dual to induction. An inductive data definition describes how the data may be built up from smaller building blocks:

$$\begin{aligned} \mathbf{List}\ A \stackrel{\text{def}}{=}\ & \texttt{cons} : (A, \mathbf{List}\ A) \to \mathbf{List}\ A \\ \mid\ & \texttt{nil} : \mathbf{List}\ A \end{aligned}$$

Here, A represents some data type. There are two *constructors*. In the first, we combine a value and a list to obtain a new list. The data can terminate through the second constructor: the empty list. The inductive way in which the definition is given allows for a deconstruction, eventually reaching some base case. This means that proving properties of inductive data is generally best done using structural induction. Consider the following statement about list length after concatenation.

Statement 1. $\forall l_1, l_2 : \texttt{length}\ l_1 + \texttt{length}\ l_2 = \texttt{length}(l_1 +\!\!+\, l_2)$

We can proof its correctness by taking $l_1 = [\,]$ as a base case, and then taking $l_1 = \texttt{cons}(x, l_1)$ for some x, l_1' as the induction step. In contrast, coinductive data definitions have no base case, which means it is possible to define infinite data structures. One of the simplest infinite data structures is known as a stream (or infinite list):

$$\mathbf{Stream}\ A \stackrel{\text{def}}{=} \texttt{head} : \mathbf{Stream}\ A \to A$$

$$\mid \texttt{tail} : \textbf{Stream}\ A \to \textbf{Stream}\ A$$

In this case the data definition is given in terms of *observations*. Note that the codomains of the list constructors is lists, while the domains of the stream observations is streams. Observations describe how to access the contents of data that is already assumed to be there, while constructors describe how to build up, or generate the data. The infinite nature of streams is reflected by the type of `tail`. The tail of a stream must always be a another stream, while a similar operation on a list has to account for the fact that at some point the empty list is reached. When proving properties of coinductive data, the proofs often also take a coinductive shape. Consider the following functions:

$$\texttt{repeat} : A \to \textbf{Stream}\ A \qquad\qquad \texttt{cycle} : \textbf{List} A \to \textbf{Stream}\ A$$

$$\texttt{interleave} : \textbf{Stream}\ A \to \textbf{Stream}\ A \to \textbf{Stream}\ A$$

Here, `repeat` takes a value and creates a stream repeating that value indefinitely. The function `cycle` takes a list of values and repeats the list indefinitely. Finally, `interleave` takes two streams and returns values from one or the other in an alternating fashion. Now consider the statement:

Statement 2. `cycle` $[0,1] =$ `interleave` (`repeat` 0) (`repeat` 1)

A proof of this statement would deconstruct the streams on either side of the equals sign, showing that their heads are identical and continuing with the tails. Note that, similar to coinductive data, coinductive proofs do not have a natural end. Instead, the truth of a statement has to be judged on the level of proofs, by noting the repetition of the proof steps.

As illustrated in the introduction, working coinductively has many advantages. A coinductive program can be more intuitive, easier to work with and easier to reason about. While our proposed language supports many different kinds of coinductive data, streams are one of the simplest forms, making them a great starting point. We therefore elaborate on the concept of streams:

Streams A stream is an infinite sequence of data, that is made available in discrete packets over time. An example is the stream of all natural numbers (without 0): `nStr` $= [1, 2, 3, \ldots$. Note that while some sources define streams to be possibly infinite, our definition requires that they are infinite. As streams are infinite, functions operating on them must produce partial results if they are to be useful. A filter function that removes all the odd numbers could not insert all the even numbers it receives into some conventional (read: finite) buffer and return said buffer, as the input data will never end. It would have to return elements piece-wise, creating another stream. A function like this (of type `Stream` $A_1 \to \ldots \to$ `Stream` A_n) is called a stream function.

Causality A stream function is causal if the n^{th} output is dependent on only the first n inputs. More formally, using $\lfloor s \rfloor_n$ as the notation for the length n prefix of s:

Definition 1 (Causality)**.** *A stream function f is causal if, for all n and all streams s and s', if $\lfloor s \rfloor_n = \lfloor s' \rfloor_n$, then $\lfloor f(s) \rfloor_n = \lfloor f(s') \rfloor_n$.*

The successor function $succ : \mathbb{N}^\omega \to \mathbb{N}^\omega$ is causal. So is a function returning a stream of sums of every two consecutive elements from the input stream. The tail function, however, is not causal. By definition, its first output is (dependent on) the second input. In our proposed semantics, causality violations will motivate the addition of new terms, to avoid typing problems for acausal functions.

2.8 Productivity

In the context of semantics, productivity (see [11], Section 8.4) refers to the following property of a program:

Definition 2 (Productivity)**.** *A program is productive if all finite sub-outputs can be computed within a finite number of steps.*

This notion can be extended to entire programming languages:

Definition 3 (Language productivity)**.** *A programming language is productive if all its valid programs are productive.*

Productivity is desirable in many settings. The estimation process example from the introduction obviously benefits from productivity. Another example of a program that needs to be productive is a control system. Such a program must run indefinitely, but is also expected to react to its environment and produce results within finite amounts of time. In terms of coinductive programming, one might think of a control system as a program that should productively calculate an infinite series of control actions to be performed. The simply typed lambda calculus is a productive language, but it is too weak to be used to implement a control system. It does not support the definition of an infinite (i.e. coinductive) process. Our proposed language does support productive coinductive definitions, where productivity is guaranteed at the level of types. It should be noted that productive coinductive computations do not yet give us Turing completeness, but it gives us more computational strength than a simply typed lambda calculus.

3 GHOPFL

We now introduce a guarded, higher order, probabilistic, functional programming language, called GHOPFL.

As we intend to define call-by-value value semantics, we firstly define values in Grammar 2. GHOPFL's terms are given by Grammar 3. Note the mutual recursion between values and terms, indicating that values may contain unevaluated terms. Let p be a metavariable ranging over $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$, representing the boolean true and false values respectively. Numerical constants (in $\mathbb{R}$) are denoted c and variables are denoted x, y. We take prev and box to have the highest precedence, followed by the single argument term-formers, like $(\mathsf{in}\ t)$. These all have the same precedence.

The basis of our language is a simply typed lambda calculus with variables, abstraction and application. We add some standard arithmetic, boolean and relative operators. Product insertion is done using the (t, s) term former and projections are available through fst and snd. Coproduct injection is done through the inL and inR term-formers. Coproduct extraction can be done using match. We include a simple if-then-else term-former, which evaluates either the term in the then, or the else part, depending on whether the boolean it takes evaluates to $\mathtt{tt}$ or $\mathtt{ff}$. The normal term-former allows us to sample any normal distribution. The term-formers (fix, in, out, next and $\circledast$) form the basis for guarded recursion, which will be elaborated on in Section 3.4. Finally, the box, unbox and prev term-formers allow us to avoid some of the problems that come with guarded recursion, explained further in Section 3.4.2.

Both the box and prev term-formers use *explicit substitution lists*, denoted with ℓ. An explicit substitution list is a list of terms $\vec{t}$ to substitute for list of variables $\vec{x}$, which we will denote $[\vec{x} \leftarrow \vec{t}]$. We will use the symbol ε for the empty substitution list and ι for the substitution list that contains identity substitutions for all the free variables in the other argument of the term-former. For example, $(\mathsf{box}\,\iota.\ t)$ is equivalent to $(\mathsf{box}\,[\vec{x} \leftarrow \vec{x}].\ t)$, with $\vec{x}$ being a list of all the free variables in t (see Section 3.1).

$$v ::= c \mid p \mid x \mid \lambda x.\ t \mid \mathsf{next}\ t \mid \mathsf{in}\ t \mid \mathsf{out}\ t \mid (t, s) \mid \mathsf{inL}\ t \mid \mathsf{inR}\ t \mid \mathsf{box}\,\ell.\ t$$

Grammar 2: Values in GHOPFL.

$$\begin{aligned} t, s, r ::= {} & v \\ \mid {} & \mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r \mid \mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \} \\ \mid {} & \mathsf{fix}\,x.\ t \mid t \circledast s \mid \mathsf{unbox}\ t \mid \mathsf{prev}\,\ell.\ t \\ \mid {} & t\ \ s \mid \mathsf{normal}\ t \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\ \mid {} & -t \mid t + s \mid t - s \mid t * s \mid t/s \\ \mid {} & t \wedge s \mid t \vee s \mid \neg p \\ \mid {} & t = s \mid t < s \end{aligned}$$

Grammar 3: Terms in GHOPFL.

3.1 Substitution

$$f_v(\mathsf{prev}\,[\vec{x} \leftarrow \vec{t}].\ t) \stackrel{\text{def}}{=} f_v(\vec{t}) \cup (f_v(t) \setminus \vec{x})$$

$$f_v(\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}) \stackrel{\text{def}}{=} f_v(t) \cup (f_v(s) \setminus \{x\}) \cup (f_v(r) \setminus \{y\})$$

$$f_v(\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r) \stackrel{\text{def}}{=} f_v(t) \cup f_v(s) \cup f_v(r) \qquad f_v(\mathsf{box}\,[\vec{x} \leftarrow \vec{t}].\ t) \stackrel{\text{def}}{=} f_v(\vec{t}) \cup (f_v(t) \setminus \vec{x})$$

$$f_v(\gamma(t_1, \ldots, t_{|\gamma|})) \stackrel{\text{def}}{=} \cup_{k \leq |\gamma|} f_v(t_k) \qquad f_v(t/s) \stackrel{\text{def}}{=} f_v(t) \cup f_v(s) \qquad f_v(c) \stackrel{\text{def}}{=} \emptyset$$

$$f_v(\mathsf{inR}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(\mathsf{unbox}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(x) \stackrel{\text{def}}{=} x$$

$$f_v(\mathsf{in}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(\mathsf{inL}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(t\ \ s) \stackrel{\text{def}}{=} f_v(t) \cup f_v(s)$$

$$f_v(\mathsf{fix}\, f.\ t) \stackrel{\text{def}}{=} f_v(t) \setminus \{f\} \qquad f_v(\mathsf{out}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v((\,t, s\,)) \stackrel{\text{def}}{=} f_v(t) \cup f_v(s)$$

$$f_v(\mathsf{normal}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(\mathsf{fst}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(\lambda x.\ t) \stackrel{\text{def}}{=} f_v(t) \setminus \{x\}$$

$$f_v(\mathsf{next}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(\mathsf{snd}\ t) \stackrel{\text{def}}{=} f_v(t) \qquad f_v(t \circledast s) \stackrel{\text{def}}{=} f_v(t) \cup f_v(s)$$

Figure 1: Free variables.

We adopt the conventional notion of capture-avoiding substitution. We define the free variables (f_v) in a GHOPFL term by structural recursion, shown in Figure 1. It should be noted that $\mathbf{fail}[s/x] = \mathbf{fail}$. Furthermore, for both ($\mathsf{box}\,[\vec{x} \leftarrow \vec{t}].\ t$) and ($\mathsf{prev}\,[\vec{x} \leftarrow \vec{t}].\ t$), $f_v(t) \setminus \vec{x} = \emptyset$.

3.2 Types

The types of GHOPFL are given by Grammar 4. We use **real** and **bool** to denote the types of the real numbers and the booleans respectively. Members of these types are in $\mathbb{R}$ and $\mathbb{B}$. There are six type formers:

- $\times$ is the type former for products.
- $+$ is the type former for co-products.
- $\to$ is the type former for functions.
- $\blacktriangleright$ is the type former for the later modality.
- $\blacksquare$ is the type former for the constant modality.
- μ is the type former for iso-recursion.

Note that, because of the type-former μ, we may also have free and bound variables at the level of types. We say that the binder μX in $\mu X.\ A$ binds occurrences of the type variable X in A. A type is closed if it contains no free type variables.

Type formation The type formation rules for GHOPFL are given in Figure 2. Here it is important to note that the $\blacksquare$ type-former is only allowed on closed types, which disallows the creation of recursive types with a recursion variable that lives under a $\blacksquare$. This is reflected by the rule for $\blacksquare$: No (potentially free) variables from Δ are in the type A (noting that $\cdot$ denotes the empty typing context).

$$A, B ::= | \mathbf{bool} | \mathbf{real} | X | A + B | A \times B | A \to B | \blacktriangleright A | \mu X.\, A | \blacksquare A$$

Grammar 4: GHOPFL's types.

$$\frac{}{\Delta \Vdash \mathbf{real} : \mathbf{Ty}} \qquad \frac{X \in \Delta}{\Delta \Vdash X : \mathbf{Ty}} \qquad \frac{\Delta, X \Vdash A : \mathbf{Ty} \quad X \text{ appears under } \blacktriangleright \text{ in } A}{\Delta \Vdash \mu X.\, A : \mathbf{Ty}}$$

$$\frac{}{\Delta \Vdash \mathbf{bool} : \mathbf{Ty}} \qquad \frac{\Delta \Vdash A : \mathbf{Ty} \quad \Delta \Vdash B : \mathbf{Ty}}{\Delta \Vdash A \times B : \mathbf{Ty}} \qquad \frac{\Delta \Vdash A : \mathbf{Ty} \quad \Delta \Vdash B : \mathbf{Ty}}{\Delta \Vdash A + B : \mathbf{Ty}}$$

$$\frac{\Delta \Vdash A : \mathbf{Ty}}{\Delta \Vdash \blacktriangleright A : \mathbf{Ty}} \qquad \frac{\cdot \Vdash A : \mathbf{Ty}}{\Delta \Vdash \blacksquare A : \mathbf{Ty}} \qquad \frac{\Delta \Vdash A : \mathbf{Ty} \quad \Delta \Vdash B : \mathbf{Ty}}{\Delta \Vdash A \to B : \mathbf{Ty}}$$

Figure 2: Type formation rules

3.3 Term formation

We introduce the term formation rules for GHOPFL in Figure 3. We call terms that adhere to these term formation rules well typed. We will consider only these terms in further discussions, such that we do not have to bother with type errors. Let $\mathcal{W}\Lambda$ denote this set of all well-typed terms. We will denote its members t, s, r. The set of all closed values is denoted $\mathcal{V}$. Its members are denoted V, W. We call $\mathcal{GV} = \mathcal{V} \cup \texttt{fail}$ generalized values and denote its members with G, H. We call a type *constant* if all occurrences of $\blacktriangleright$ are beneath a $\blacksquare$.

Notable are the types of the in and out terms. They form the isomorphism between a recursive type and an unrolling thereof, giving us iso-recursive types. For more information on recursive types, see [19].

The normal term-former brings us into the probabilistic programming paradigm by allowing us to sample any normal distribution. It takes a pair of real numbers as argument, representing the mean and variance of the distribution to be sampled. We only include the sampling of normal distributions, but our results should hold for all continuous distribution sampling terms.

3.3.1 Base type operations

We introduce a generalized set of terms acting on base types to be able to compactly define the semantics of operations such as arithmetic addition and logical conjunction. Let $\mathcal{B} = \{\mathbf{real}, \mathbf{bool}\}$ denote the set of base types. Let Σ be a set of *signatures* $\gamma : \tau \to \beta$, where $\tau = (\beta_1, \ldots, \beta_n) \in \mathcal{B}^n$ with $n \geq 1$ and $\beta \in \mathcal{B}$. Let Σ contain the signatures for all arithmetic, boolean and relative operators except division. We will handle division separately because of the division by zero complications. We use / to denote the division term-former and use $\div : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ to denote its implementation.

$$\frac{\Gamma \vdash t : \mathbf{real} \quad \Gamma \vdash s : \mathbf{real}}{\Gamma \vdash t/s : \mathbf{real}} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : \blacktriangleright(A \to B) \quad \Gamma \vdash s : \blacktriangleright A}{\Gamma \vdash t \circledast s : \blacktriangleright B} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\ t : \blacktriangleright A}$$

$$\frac{\Gamma \vdash t : A[\mu X.\ A/X]}{\Gamma \vdash \mathsf{in}\ t\ s : B} \qquad \frac{\Gamma, x : \blacktriangleright A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.\ t : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ \ s : B}$$

$$\frac{\Gamma \vdash t : \mu X.\ A}{\Gamma \vdash \mathsf{out}\ t\ s : A[\mu X.\ A/X]} \qquad \frac{\Gamma \vdash s : C \quad \Gamma \vdash r : C \quad \Gamma \vdash t : A + B}{\Gamma \vdash \mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \} : C} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash s : B}{\Gamma \vdash (t, s) : A \times B}$$

$$\frac{\Gamma \vdash t : \blacksquare A}{\Gamma \vdash \mathsf{unbox}\ t : A} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inL}\ t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inR}\ t : A + B} \qquad \frac{\Gamma \vdash t : \mathbf{bool} \quad \Gamma \vdash s : A \quad \Gamma \vdash r : A}{\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r : A}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\ t : A} \qquad \frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : \blacktriangleright A \quad \Gamma \vdash t_1 : A_1, \ldots, t_n : A_n}{\Gamma \vdash \mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t : A} \quad A_1 \ldots A_n \text{ constant}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\ t : B} \qquad \frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : \blacktriangleright A \quad \Gamma \vdash t_1 : A_1, \ldots, t_n : A_n}{\Gamma \vdash \mathsf{box}\ [\vec{x} \leftarrow \vec{t}].\ t : \blacksquare A} \quad A_1 \ldots A_n \text{ constant}$$

$$\frac{\gamma : (\beta_{\mathbf{1}}, \ldots, \beta_{|\gamma|}) \to \beta \in \Sigma \quad \Gamma \vdash t_1 : \beta_1, \ldots, t_{|\gamma|} : \beta_{|\gamma|}}{\Gamma \vdash \gamma(t_1, \ldots t_{|\gamma|}) : \beta} \qquad \frac{}{\Gamma \vdash \mathrm{normal} : \mathbf{real} \times \mathbf{real} \to \mathbf{real}}$$

Figure 3: Term formation rules

Let $\overline{\mathbf{real}} = \mathbb{R}$ and $\overline{\mathbf{bool}} = \mathbb{B}$. Now let $\overline{\tau} = \prod_{1 \le i \le n} \overline{\beta_i}$. We denote the implementation of γ by $\sigma_\gamma : \overline{\tau} \to \overline{\beta}$. Finally, let $|\gamma| = n$ denote the arity of γ.

The remaining terms all have to do with recursion. They work together in a non-trivial way to give us productive guarded recursion, or to alleviate some of the restrictions thereof. The following section will describe how this is achieved.

3.4 Recursion

Recursion is problematic when it comes to productivity guarantees. It is clear to see that without proper care, a programmer may write programs that will never produce results. We use the notion of the *later modality* in our type system to combat this. The later modality was originally introduced by Nakano [18] and has formed the basis for many co-programming systems since. It distinguishes data we have access to immediately from data we will have access to at a later moment of evaluation. The type-former $\blacktriangleright$ embeds a type into this modality and the term-former next creates a term of this type. To work with this modality, we use an unconventional fixpoint operator. Traditionally, a fixpoint operator is of the type $(A \to A) \to A$. Using this kind of fixpoint operator, we can write

recursive functions like the factorial function:

$$\texttt{Fac} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * (f\ \ (n-1)) \quad : \quad \mathbf{real} \to \mathbf{real}$$

During evaluation of `Fac`, f would be substituted by a new copy of `Fac`. In doing so, the fix operator effectively uses the function $f : \mathbf{real} \to \mathbf{real}$ to calculate a result of type **real**. This kind of fixpoint operator can be problematic when it comes to termination. Consider the fixpoint for a successor function:

$$\mathsf{fix}\, f.\ \lambda n.\ f\ \ (n+1) \quad : \quad \mathbf{real} \to \mathbf{real}$$

Once again, f has the type $\mathbf{real} \to \mathbf{real}$, but this time it is clear to see that the fixpoint operator will get stuck infinitely applying f to higher and higher n.

To avoid this problem, we define a fixpoint operator with type $(\blacktriangleright A \to A) \to A$. This fixpoint operator shows the guarded self-referencing, required by the type-formation rule for μ: The function that is applied repeatedly to obtain a fixpoint must take a guarded instance of A to create another instance of A. This means a $\blacktriangleright$ must be introduced for each step in the fixpoint calculation, ensuring that evaluation may be stopped at every step. The concrete mechanisms through which this operator works are illustrated in Section 3.4.1. Note that under this new operator, it is no longer possible to define recursive functions on data types like **real**, as there is no way to create a **real** from a $\blacktriangleright$ **real**.

The term formation rule for fix tells us that the recursion variable for a recursive function of return type A must be of type $\blacktriangleright A$, meaning that deeper and deeper recursively calculated results must be embedded in more and more $\blacktriangleright$. Note that we may still work with delayed values through the next term-former (artificially delaying a value) and the $\circledast$ term-former allowing for the application of delayed functions. Intuitively, if we have function that becomes available later and a value in its domain that also becomes available later, we may apply the function to the value to obtain a delayed result. Note that we may also use term-formers on delayed values, as we can simply write functions that apply the desired term-formers to their arguments and then use $\circledast$.

3.4.1 Guarded streams

Using the later modality, we can define guarded streams of type A with the recursive type definition:

$$\mathbf{gStr}\ A \stackrel{\text{def}}{=} \mu X.\ A \times \blacktriangleright X$$

The guardedness manifests itself in that the head of the stream is available now, but any calculations on the tail must happen under the later modality. A guarded version of the stream of natural numbers (approximated with real numbers, as we do not have an integer type), as described earlier, may be defined as follows:

$$(\mathsf{fix}\, f.\ \lambda x.\ \mathsf{in}\ (\ x, f \circledast \mathsf{next}\ (x + 1.0)\ ))\ \ 1.0$$

Here, f is of the type $\blacktriangleright(\mathbf{real} \to \mathbf{gStr}\,(\mathbf{real}))$. Thus, by the term-formation rule for $\circledast$, we derive that $(f \circledast \mathsf{next}\ (x+1))$ is of the type $(\blacktriangleright\, \mathbf{gStr}\,(\mathbf{real}))$. Now, using product insertion in combination with the in term-former we can take this $(\blacktriangleright\, \mathbf{gStr}\,(\mathbf{real}))$ and create another instance of $\mathbf{gStr}\,(\mathbf{real})$. Doing this repeatedly gives us a fixpoint operator of the expected signature:

$$(\blacktriangleright\, \mathbf{gStr}\,(\mathbf{real}) \to \mathbf{gStr}\,(\mathbf{real})) \to \mathbf{gStr}\,(\mathbf{real})$$

In general, if the data type A has a constructor of the shape $(\cdots \to \blacktriangleright A \to A)$, this fixpoint operator can be used to create well-typed, guarded recursion. In this case, we have such a constructor:

$$\texttt{gCons} \stackrel{\text{def}}{=} \lambda x.\, \lambda s.\, \mathsf{in}\,(x, s) \quad : \quad A \to \blacktriangleright \mathbf{gStr}\, A \to \mathbf{gStr}\, A$$

Examples of other guarded recursive coinductive data we can create are:

- Infinite binary trees: $\mu X.\, A \times \blacktriangleright(X \times X)$.
- The conatural numbers: $\mu X.\, B + \blacktriangleright X$.
- Colists: $\mu X.\, B + \blacktriangleright(A \times \blacktriangleright X)$.

Here B is ideally some singleton type, but we can also use some base type. We can define some other common stream operations on guarded streams:

$$\texttt{gHead} \stackrel{\text{def}}{=} \lambda s.\, \mathsf{fst}\ \mathsf{out}\ s \qquad\qquad \texttt{gTail} \stackrel{\text{def}}{=} \lambda s.\, \mathsf{snd}\ \mathsf{out}\ s$$

Here, it is important to note that `gHead` is of type $\mathbf{gStr}\, A \to A$, while `gTail` is of type $\mathbf{gStr}\, A \to \blacktriangleright \mathbf{gStr}\, A$. Every successive application of `gTail` introduces another $\blacktriangleright$.

With this kind of stream, using only the later modality, we run into some restrictions. For example, a function that returns a stream containing every second element of the input stream is not typable, as it not *causal* (see Section 2.7). A first attempt might look like:

$$\texttt{every2nd} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\, \lambda s.\, \mathsf{in}\,(\,\texttt{gHead}\ s, f \circledast \texttt{gTail}\ (\texttt{gTail}\ s)\,)$$

Here, the right operand of $\circledast$ should be of type $\blacktriangleright \mathbf{gStr}\, A$, but (`gTail` (`gTail` s)) is of type $\blacktriangleright\blacktriangleright \mathbf{gStr}\, A$. We need some way to remove the $\blacktriangleright$, without getting rid of our productivity guarantees, so that we may write acausal stream functions, like `every2nd`.

3.4.2 Boxed recursive types

A second modality that grants us a bit more freedom when it comes to working with coinductive types, was introduced by Clouston et al. [8]. It is named the constant modality and allows us to build coinductive types with less restrictions than the guarded recursive types we have seen so far. The type-former $\blacksquare$ embeds a type into the constant modality. Using the term-former box, we can create a term of this type. Its inverse, unbox, removes the $\blacksquare$. Data can be marked *constant* using the box type-former, allowing it be be used in a prev term-former. Using prev, we can remove instances of the later modality ($\blacktriangleright$) in a controlled manner. Constraining prev to only work with constant types allows us to gain access to delayed data without losing productivity, as any data that is truly delayed (contains a $\blacktriangleright$ without a $\blacksquare$ above it) cannot have its time displacement removed. Adding the same constant constraint to the box term-former enforces the intuitive notion that terms of constant type must consist of constant type parts. We can use `prev` to remove a later modality in the following example:

$$\mathsf{prev}\, \varepsilon.\, (\mathsf{next}\ 1.0) : \mathbf{real}$$

Here, the second argument of prev contains no free variables and can therefore definitely be interpreted as something that is available now, as there is no way to write a non-terminating program

without variables. Even when the second argument contains free variables, we can safely remove the later modality if the variables are constant:

$$(\lambda x.\ \mathsf{prev}\,\iota.\ \mathsf{next}\ f \circledast \mathsf{next}\ x)\ y$$

Here f is some unspecified function of constant type: $(A \to B)$. This means $x : A$ is constant, by which we know that the only true delay we are introducing onto it, is introduced inside of the prev, meaning we can safely remove it.

Note that this program requires that x be identity substituted, meaning that the x in the lambda is not the same as the one in the delayed application. Instead, $(\mathsf{prev}\,\iota.\ \ldots)$ is short for $(\mathsf{prev}\,[x \leftarrow x].\ \ldots)$, where the second x is the one bound by the lambda, and the first x binds the x in the delayed application. The necessity of this substitution step is best explained in a denotational setting, but it may be understood intuitively as follows: The constant types exist all at once and live in their own world, separate from the time-displaced types. As $(\mathsf{next}\ f \circledast \mathsf{next}\ x)$ is time-displaced, while the x bound by the λ is constant, we have to inject this x into the time-displaced regime. In order to keep programs readable, we often substitute x for a variable of the same name, but which now lives in the time-displaced world.

To see where the limitations around prev are, that cause it not to break productivity guarantees, we now give a situation where y is not constant:

$$(\lambda x.\ \mathsf{prev}\,\iota.\ \mathsf{next}\ f \circledast x)\ y$$

This program is not well-typed: By the term-formation rule for $\circledast$, x must be of type $\blacktriangleright A$, but by the term-formation rule for prev, we know that x must be constant, giving us a contradiction. Intuitively, we do not know exactly how much delay x is under, meaning that we cannot make sure that the function f may be applied. Note that it is not the type of the argument that needs to be constant, but the types of the free variables within it. This restriction also prevents us from directly negating the delay that is introduced by a recursion step like shown below:

$$\mathsf{fix}\,r.\ \lambda x.\ \mathsf{prev}\,\iota.\ r \circledast \mathsf{next}\ (x)$$

While the delay imposed on x is imposed within the prev, by the term formation rules, the recursion variable r must be of type $\blacktriangleright(A \to B)$, making this program ill-typed. To show how the box term-former allows us to leverage the powers of prev in the context of guarded recursive types, we introduce boxed streams.

3.4.3 Boxed streams

Using $\blacksquare$, we can lift guarded recursive streams to boxed streams. We define the latter in terms of the former:

$$\mathbf{bStr}\ A \stackrel{\text{def}}{=} \blacksquare\,\mathbf{gStr}\ A$$

For this kind of stream, we can use unbox and prev to define the following functions:

$$\mathtt{bHead} \stackrel{\text{def}}{=} \lambda s.\ \mathtt{gHead}\ (\mathsf{unbox}\ s) \qquad \mathtt{bTail} \stackrel{\text{def}}{=} \lambda s.\ \mathsf{box}\,\iota.\ \mathsf{prev}\,\iota.\ \mathtt{gTail}\ (\mathsf{unbox}\ s)$$

$$\mathtt{bCons} \stackrel{\text{def}}{=} \lambda e.\ \lambda s.\ \mathsf{box}\,\iota.\ \mathsf{in}\ (\ e, \mathsf{next}\ \mathsf{unbox}\ s\ )$$

Note that we still have a constructor (`bCons`) with the correct type signature for our fixpoint operator. Here, the type of `bTail` crucially lacks the later modality: $\mathbf{bStr}\, A \to \mathbf{bStr}\, A$. This is achieved by the prev term-former. Its argument contains just one free variable: $s : \mathbf{bStr}\, A$, which is constant. When we unbox this variable, we obtain a guarded recursive stream, to which we can apply the guarded recursive head and tail functions. This time, however, we may use `prev` after extracting the tail. The resulting guarded stream must then be re-boxed. Note that $\mathbf{bStr}\, A$ for non constant A is ill-defined, as there is no closed term that describes a guarded stream for such a type (contrary to the example of the guarded stream of natural numbers).

Using `bTail`, we can easily write functions that skip any number of elements in the stream, without any mention of $\blacktriangleright$ (i.e. still having type: $\mathbf{bStr}\, A \to \mathbf{bStr}\, A$). It is clear that functions such as `every2nd` are now implementable. In fact, the family of functions below takes every n-th value from the stream.

$$\texttt{bDrop}_n \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda s.\ \texttt{bCons}\ (\texttt{bHead}\ s)\ (f \circledast \mathsf{next}\, (\overbrace{\texttt{cTail}\ \ldots\ \texttt{bTail}}^{n}\ s))$$

It should be noted that it is not possible to parametrise `cDrop` with the amount of elements to drop, as the recursion needed for such a function would still introduce time displacements. Furthermore, it should be emphasized that we can not just box anything we want to and proceed to remove its later modalities. We are able to box a guarded stream of constant elements, as it may be constructed using a closed term of constant type.

At this point, we have given and motivated the syntax for GHOPFL. We will now give the semantics for this language, so that we may see how its programs evaluate.

4 Small-step semantics

We define the main computation steps of the small-step semantics of GHOPFL in terms of a reduction relation. Additionally, we introduce a family of small-step relations that include the reduction relation, as well as some rules regarding the next term-former and the sampling of normal distributions. Finally, we define a family of multi-step relations as the reflexitive transitive closure of the small-step relations. The multi-step relation then forms the call-by-value small-step semantics for GHOPFL. These small step semantics consist of very simple rules, making them very useful for theoretically reasoning about properties of GHOPFL. As no particular order of evaluation (other than call-by-value) is enforced by these semantics, there may be multiple ways to reduce a program. As we will mention later, confluence can be guaranteed meaning that all reductions should give the same results.

To avoid having to specify how errors are handled in every situation, we define our reduction relation to take place within evaluation contexts. Evaluation contexts are given by Grammar 5. There is always just one hole, denoted by $[\cdot]$, in any evaluation context. We use the notation $C[t]$ to mean $C[t/[\cdot]]$ in terms of regular substitution. As will become clear later, we may not always evaluate into next terms. Because of this, there is no context for this term-former meaning that failures must be handled separately.

$$
\begin{aligned}
C ::= {} & [\cdot] \mid \mathsf{fst}\ C \mid \mathsf{snd}\ C \mid \mathsf{out}\ C \\
\mid {} & (\lambda x.\ t)\ C \mid C\ t \mid C \circledast t \mid t \circledast C \mid \mathsf{unbox}\ C \mid \mathsf{prev}\ C \\
\mid {} & \mathsf{if}\ C\ \mathsf{then}\ t\ \mathsf{else}\ s \mid \mathsf{match}\ C\ \{\ \mathsf{inL}\ x \mapsto t\ ;\ \mathsf{inR}\ y \mapsto s\ \} \\
\mid {} & \gamma(t_1, \ldots, t_{i-1}, C, t_{i+1} \ldots, t_{|\gamma|}) \quad 1 \leq i \leq |\gamma|
\end{aligned}
$$

Grammar 5: Evaluation contexts.

We define a reduction relation $\twoheadrightarrow$ within contexts. It is shown in Figure 4. Note that division by zero is not defined in this relation. The call-by-value strategy is enforced by (Red App) only allowing reductions when the right-hand side is a value, in combination with the presence of the context $((\lambda x.\ t)\ C)$. The notion of delayed function application is illustrated in (Red DApp). The utility of contexts in error handling is shown in (Red CFail), which implicitly creates rules for propagating up **fail** terms for all the term-formers in Grammar 5.

We define a family of small-step relations. It is indexed on depth ($\rightarrow_n,\ n \in \mathbb{N}$), to disallow calculations past a given number of "next" terms. The relation $\rightarrow_n$ may evaluate into n next terms, before evaluation stops. This indexation approach was inspired by Abel and Vezzosi's method of defining a similar reduction relation [1]. The rules for these small-step relations are shown in Figure 5. We make use of Dexter Kozen's insight that a probabilistic program can be evaluated as a deterministic program, parametrised by a list of random draws [15]. We will denote such lists S. We will keep track of the density of an execution indeed sampling the values in S and use w to denote such densities. Note that we must consider densities rather than chances as we are dealing with continuous distributions. We use $\mathrm{PDF}_{(\mu,\sigma)}$ to denote the probability density function of a Gaussian distribution with mean μ and variance σ. The reduction relation is part of all of the small-step relations and does not change S or w.

$$C[(\lambda x.\ t)\ V] \twoheadrightarrow C[t[V/x]] \quad \text{(Red App)} \qquad C[\gamma(\beta_1, \ldots, \beta_{|\alpha|})] \twoheadrightarrow C[\sigma_\gamma(\overline{\beta_1}, \ldots, \overline{\beta_{|\alpha|}})] \quad \text{(Red Sig)}$$

$$C[\mathsf{fst}\ (t, s)] \twoheadrightarrow C[t] \quad \text{(Red Fst)} \qquad C[\mathsf{fix}\, f.\ t] \twoheadrightarrow C[t[\mathsf{next}\ (\mathsf{fix}\, f.\ t)/f]] \quad \text{(Red Fix)}$$

$$C[\mathsf{snd}\ (t, s)] \twoheadrightarrow C[s] \quad \text{(Red Snd)} \qquad C[\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ t\ \mathsf{else}\ s] \twoheadrightarrow C[t] \quad \text{(Red IfTrue)}$$

$$C[\mathsf{out}\ (\mathsf{in}\ t)] \twoheadrightarrow C[t] \quad \text{(Red Out)} \qquad C[\mathsf{if}\ \mathtt{ff}\ \mathsf{then}\ t\ \mathsf{else}\ s] \twoheadrightarrow C[s] \quad \text{(Red IfFalse)}$$

$$C[\mathbf{fail}] \twoheadrightarrow \mathbf{fail} \quad \text{(Red CFail)} \qquad C[\mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t] \twoheadrightarrow C[\mathsf{prev}\ t[\vec{t}/\vec{x}]] \quad \text{(Red Prev)}$$

$$\mathsf{next}\ \mathbf{fail} \twoheadrightarrow \mathbf{fail} \quad \text{(Red NFail)} \qquad C[\mathsf{next}\ t \circledast \mathsf{next}\ s] \twoheadrightarrow C[\mathsf{next}\ (t\ s)] \quad \text{(Red DApp)}$$

$$C[\mathsf{prev}\ (\mathsf{next}\ t)] \twoheadrightarrow C[t] \quad \text{(Red PrevE)} \qquad C[c_1/c_2] \twoheadrightarrow C[c_1 \div c_2] \quad (c_2 \neq 0) \quad \text{(Red Div)}$$

$$C[\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}] \twoheadrightarrow C[s[V/x]] \quad \text{(Red MatchL)}$$

$$C[\mathsf{match}\ (\mathsf{inR}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}] \twoheadrightarrow C[r[V/y]] \quad \text{(Red MatchR)}$$

$$C[\mathsf{unbox}\ (\mathsf{box}\, [\vec{x} \leftarrow \vec{t}].\ t)] \twoheadrightarrow C[t[\vec{t}/\vec{x}]] \quad \text{(Red Box)}$$

Figure 4: Reduction rules.

It should be noted that confluence of $\to_n$ is not guaranteed under small enough n. For example, under the relation $\to_0$, the term $(\lambda x.\ \mathsf{next}\ x)\ (\mathsf{out}\ (\mathsf{in}\ y))$ reduces to two different values: $(\mathsf{next}\ y)$ and $(\mathsf{next}\ (\mathsf{out}\ (\mathsf{in}\ y)))$, depending on whether (Red App) was applied before or after (Red Out). When evaluating a program, we trust a sufficiently large n is chosen.

Note that many programs that use the prev term-former can be totally evaluated under $\to_0$, as all $\blacktriangleright$ occurrences are removed before they need to be evaluated under. We keep $\to$ indexed to ensure productivity guarantees for programs that do not involve prev.

$$\frac{(t, w, S) \to_n (t', w, S)}{(\mathsf{next}\ t, w, S) \to_{n+1} (\mathsf{next}\ t', w, S)} \quad \text{(Small next)} \qquad \frac{t \twoheadrightarrow t'}{(t, w, S) \to_n (t', w, S)} \quad \text{(Small Red)}$$

$$\frac{w' = \mathrm{PDF}_{(\mu,\sigma)}(c) \quad w' > 0}{(C[\mathsf{normal}\ (\mu, \sigma)], w, c : S) \to_n (C[c], w \cdot w', S)} \quad \text{(Small Norm)}$$

$$\frac{\mathrm{PDF}_{(\mu,\sigma)}(c) = 0}{(C[\mathsf{normal}\ (\mu, \sigma)], w, c : S) \to_n (C[\mathbf{fail}], 0, S)} \quad \text{(Small Fail)}$$

Figure 5: The rules for small step semantics.

Redexes We call the terms for which there exists a small-step rule, like projections of pairs and $(\mathsf{normal}\ (\mu, \sigma))$, redexes. They are denoted R. We call a term reducible if it can be written as $C[R]$. Note that for the relation $\to_0$, the term $\mathsf{next}\ t$ is not a redex.

We now proof some properties of the small-step relation, that will aid us in proving equivalence of the small-step and big-step semantics.

Lemma 1 (Small-step list composition)**:**
If $(t, w, S) \to_n (t', w', S')$*, then for any* S^**,* $(t, w, S +\!\!+ S^*) \to_n (t', w', S' +\!\!+ S^*)$

Proof. By case analysis on the derivation of $(t, 1, S) \to_n (t', w', S')$:

Case 1.1 (Small Red, Small next)
Here the draws list is unaltered, and so the desired result follows trivially.

Case 1.2 (Small Norm)
By assumption: $S = c :: S_0$ for some c, S_0. Then, $S +\!\!+ S' = (c :: S_0) +\!\!+ S' = c :: S_0 +\!\!+ S'$. Now the result follows immediately from (Small Norm).

Case 1.3 (Small Fail)
Analogous to Case 1.2

All cases hold, and thus Lemma 1 holds. □

Lemma 2 (Small-step density composition)**:**
If $(t, w, S) \to_n (t', w', S')$*, then for any* $w^* \geq 0$*,* $(t, w \cdot w^*, S) \to_n (t', w' \cdot w^*, S')$

Proof. By case analysis on the derivation of $(t, w, S) \to_n (t', w', S')$:

Case 2.1 (Small Red, Small Norm)
Here w is unaltered, and so the desired result follows trivially.

Case 2.2 (Small Norm)
Here, $w' = w * w_0$ for some w_0. By associativity and commutativity of $(\cdot)$: $(w \cdot w^*) \cdot w_0 = (w \cdot w_0) \cdot w^*$.

Case 2.3 (Small Fail)
Here $w' = 0$, and thus $w' \cdot w^* = 0$.

All cases hold and thus Lemma 2 holds. □

Lemma 3 (Nested contexts are contexts)**:**
$C[C'[t]] = C''[t]$*, for the context* $C'' = C[C']$*.*

Proof. Let the derivation of C be the sequence of productions $P_1^C \dots P_n^C$. Let the derivation of C' be the sequence of productions $P_1^{C'} \dots P_k^{C'}$. As all productions except $C \to [\cdot]$ introduce precisely one C, P_n^C and $P_k^{C'}$ must be $C \to [\cdot]$. Because the $[\,]$ operator is defined as substitution of $[\cdot]$, $C[C']$ can be derived by the productions $P_1^C \dots P_{n-1}^C, P_1^{C'} \dots P_k^{C'}$. Note that there still only exists one hole. □

Lemma 4 (Context independent reduction)**:**
If $C[t] \twoheadrightarrow C[t']$ *for some context* C*, then for all* C'*,* $C'[t] \twoheadrightarrow C'[t']$*.*

Proof. Reduction within contexts is defined independent from the structure of the context and for any $C[r] \twoheadrightarrow C[r']$, r' is uniquely determined by r. □

Lemma 5 (Reduction within contexts)**:**
If $t \twoheadrightarrow t'$ *and there is no* C' *such that* $t = C'[\textbf{\textit{fail}}]$ *and* $t \neq \mathsf{next}\ \textbf{\textit{fail}}$*, then for any* C*:* $C[t] \twoheadrightarrow C[t']$

Proof. By assumption, $C'[s] \twoheadrightarrow C'[s']$ for some C', s, s', with $t = C'[s]$ and $t' = C'[s']$. Then by Lemma 3, $C[C'[s]] = C''[s]$ for some C''. By Lemma 4, $C''[s] \twoheadrightarrow C''[s']$, which implies $C[t] \twoheadrightarrow C[t']$. □

Lemma 6 (Reduction outside contexts)**:**
If $C[t] \twoheadrightarrow C[t']$ for some C, then $t \twoheadrightarrow t'$.

Proof. By Lemma 4, taking $C' = [\cdot]$, $C'[t] \twoheadrightarrow C'[t'] \implies t \twoheadrightarrow t'$. □

Lemma 7 (Small-step within contexts)**:**
If $(t, w, S) \to_n (t', w', S')$ and there is no C' such that $t = C'[\textbf{\textit{fail}}]$ and $t \neq \mathsf{next}\ \textbf{\textit{fail}}$, then $(C[t], w, S) \to_n (C[t'], w', S')$

Proof. By case analysis on the derivation of $(t, w, S) \to_n (t', w', S')$:

Case 7.1 (Small Red)
Here $t \twoheadrightarrow t'$ and thus by Lemma 5, $C[t] \twoheadrightarrow C[t']$. Then by (Small Red): $(C[t], w, S) \to_n (C[t'], w', S')$

Case 7.2 (Small Norm)
Here $t = C'[\mathsf{normal}\ (\mu, \sigma)]$, $t' = C'[c]$, $S' = \mathsf{tail}\, S$ and $w' = w \cdot \mathrm{PDF}_{(\mu, sigma)}(c)$. Then by Lemma 3 and (Small Norm) we get $(C[t], w, s) \to_n (C[t'], w', S')$

Case 7.3 (Small Fail)
Analogous to Case 7.2

All cases hold and thus Lemma 7 holds. □

Lemma 8 (Small-step outside contexts)**:**
If $(C[t], w, S) \to_n (C[t'], w', S')$, then $(t, w, S) \to_n (t', w', S')$.

Proof. By case analysis on the derivation of $(C[t], w, S) \to_n (C[t'], w', S')$.

Case 8.1 (Small Red)
Here, $C[t] \twoheadrightarrow C[t']$ and thus by Lemma 6, $t \twoheadrightarrow t'$. Then by (Small Red) $(t, w, S) \to_n (t', w', S')$.

Case 8.2 (Small Norm)
Here, $t = C[\mathsf{normal}\ (\mu, \sigma)]$, $t' = C[c]$, $w' = w \cdot \mathrm{PDF}_{(\mu,\sigma)}$ with $\mathrm{PDF}_{(\mu,\sigma)} > 0$ and $S' = \mathsf{tail}\, S$. Thus, with (Small Norm), taking $C = [\cdot]$, we obtain the desired result.

Case 8.3 (Small Fail)
Analogous to Case 8.2

All cases hold and thus Lemma 8 holds. □

4.1 Multi-step

We define a family of multi-step reduction relations: For all n, $\Rightarrow_n$ is the reflexitive transitive closure of $\to_n$. We now extend the proofs from the previous section to the multi-step relation and proof some additional properties.

Lemma 9 (Multi step within next)**:**
If $(t, w, S) \Rightarrow_n (t', w', S')$, then $(\mathsf{next}\ t, w, S) \Rightarrow_{n+1} (\mathsf{next}\ t', w', S')$.

Proof. By repeated application of (Small next). □

Lemma 10 (Multi-step within contexts)**:**
If $(t, w, S) \Rightarrow_n (t', w', S')$ *and there is no* C' *such that* $t = C'[fail])$ *and* $t \neq \mathsf{next}\ \boldsymbol{fail}$*, then for any* C*,* $(C[t], w, S) \Rightarrow_n (C[t'], w', S')$

Proof. By induction on the derivation, with appeal to Lemma 7 □

Lemma 11 (Multi-step list composition)**:**
If $(t, w, S) \Rightarrow_n (t', w', S')$*, then for any* S^**,* $(t, w, S +\!\!+ S^*) \Rightarrow_n (t', w', S' +\!\!+ S^*)$

Proof. By induction on the derivation:

Base case: Reflexivity.

Induction step: Assume $(t, w, S) \Rightarrow_n (t'', w'', S'') \rightarrow_n (t', w', S')$ for some (t'', w'', S''). Then:

$$\text{For any } S^*,\ (t, w, S +\!\!+ S^*) \Rightarrow_n (t'', w'', S'' +\!\!+ S^*) \qquad \text{(By induction hypothesis)} \qquad (1)$$

$$\text{For any } S^*, (t, w, S +\!\!+ S^*) \Rightarrow_n (t', w', S' +\!\!+ S^*) \qquad \text{(By Lemma 1)} \qquad (2)$$

Hence Lemma 11 holds by induction on the derivation. □

Lemma 12 (Multi-step density composition)**:**
If $(t, w, S) \Rightarrow_n (t', w', S')$*, then for any* $w^* \geq 0$*,* $(t, w \cdot w^*, S) \Rightarrow_n (t', w' \cdot w^*, S')$

Proof. Analogous to Lemma 11, with appeal to Lemma 2. □

4.2 Example evaluation

To show how the multi-step relation assigns meanings to programs, we will now evaluate some programs involving a stream of increasing numbers:

$$\mathtt{iS} \stackrel{\text{def}}{=} (\mathsf{fix}\, f.\ \lambda x.\ \mathsf{in}\ (\ x, f \circledast \mathsf{next}\ (x+1)\ ))$$

To showcase how the distribution sampling works, we give the stream a random initial value: $\mathtt{r} \stackrel{\text{def}}{=} \mathsf{normal}\ (\ 0, 1\ )$. We will use the random draws list $[0.5]$, meaning that we expect our stream to be $(0.5, 1.5, \ldots)$. We start by extracting the first value from the stream using $(\mathsf{fst}\ \mathsf{out})$, under $\Rightarrow_0$. Note that $\mathrm{PDF}_{(0,1)}(0.5) = 0.3521\ldots$, which we will round to 0.35 for brevity.

$$((\mathsf{fst}\ \mathsf{out}\ ((\mathsf{fix}\, f.\ \lambda x.\ \mathsf{in}\ (\ x, f \circledast \mathsf{next}\ (x+1)\ ))\ \mathtt{r})), 1, [0.5]) \qquad (1)$$

$$\Rightarrow_0\ ((\mathsf{fst}\ \mathsf{out}\ ((\lambda x.\ \mathsf{in}\ (\ x, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (x+1)\ ))\ \mathtt{r})), 1, [0.5]) \qquad \text{(Red Fix)} \qquad (2)$$

$$\Rightarrow_0\ ((\mathsf{fst}\ \mathsf{out}\ ((\lambda x.\ \mathsf{in}\ (\ x, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (x+1)\ ))\ 0.5)), 0.35, [\,]) \qquad \text{(Small Norm)} \qquad (3)$$

$$\Rightarrow_0\ ((\mathsf{fst}\ \mathsf{out}\ (\mathsf{in}\ (\ 0.5, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (0.5+1)\ ))), 0.35, [\,]) \qquad \text{(Red App)} \qquad (4)$$

$$\Rightarrow_0\ ((\mathsf{fst}\ (\ 0.5, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (0.5+1)\ )), 0.35, [\,]) \qquad \text{(Red Out)} \qquad (5)$$

$$\Rightarrow_0\ (0.5, 0.35, [\,]) \qquad \text{(Red Fst)} \qquad (6)$$

Here, $\mathsf{normal}\ (0, 1)$ must be evaluated in 3 before the function application in 4 as it is not a value. Also note the changes to the densities and the draws list in this step. Now let us try to get at the second value in the stream using $(\mathsf{snd\ out})$, in combination with $(\mathsf{next}\ (\lambda x.\ \mathsf{fst\ out}\ x) \circledast \ldots)$. As the evaluation starts with the same five steps as the previous example, they will not be shown.

$$((\mathsf{next}\ (\lambda x.\ \mathsf{fst\ out}\ x) \circledast (\mathsf{snd\ out}\ (\mathtt{iS}\ \mathtt{r})), 1, [0.5]) \tag{7}$$

$$\Rightarrow_0\ ((\mathsf{next}\ (\lambda x.\ \mathsf{fst\ out}\ x) \circledast (\mathsf{snd}\ (\ 0.5, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (0.5 + 1)\ ))), 0.35, [\,]) \quad \text{(Multiple steps)} \tag{8}$$

$$\Rightarrow_0\ ((\mathsf{next}\ (\lambda x.\ \mathsf{fst\ out}\ x) \circledast (\mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (0.5 + 1))), 0.35, [\,]) \quad \text{(Red Snd)} \tag{9}$$

$$\Rightarrow_0\ ((\mathsf{next}\ (\lambda x.\ \mathsf{fst\ out}\ x) \circledast \mathsf{next}\ (\mathtt{iS}\ \ (0.5 + 1))), 0.35, [\,]) \quad \text{(Red DApp)} \tag{10}$$

$$\Rightarrow_0\ ((\mathsf{next}\ ((\lambda x.\ \mathsf{fst\ out}\ x)\ \ (\mathtt{is}\ \ (0.5 + 1)))), 0.35, [\,]) \quad \text{(Red DApp)} \tag{11}$$

At this point, the evaluation under $\Rightarrow_0$ halts, as the applications under the next can not be accessed. If we had instead evaluated the same program under $\Rightarrow_1$, we could have evaluated further, like shown below. Here, 13 just evaluates a recursion step of the function, which is done using the same steps as in the original unfolding of the recursion and is thus not shown. Furthermore, the steps taken in 15 are analogous to the ending of the first example and are also not shown.

$$((\mathsf{next}\ ((\lambda x.\ \mathsf{fst\ out}\ x)\ \ (\mathtt{iS}\ \ (0.5 + 1)))), 0.35, [\,]) \quad \text{(Under Small next)} \tag{12}$$

$$\Rightarrow_1\ ((\mathsf{next}\ ((\lambda x.\ \mathsf{fst\ out}\ x)\ \ (\mathsf{in}\ (\ 1.5, \mathsf{next}\ \mathtt{iS} \circledast \mathsf{next}\ (x + 1)\ )))), 0.35, [\,]) \quad \text{(Multiple steps)} \tag{13}$$

$$\Rightarrow_1\ ((\mathsf{next}\ (\mathsf{fst\ out}\ (\mathsf{in}\ (\ 1.5, \mathsf{next}\ \mathbf{f} \circledast \mathsf{next}\ (x + 1)\ )))), 0.35, [\,]) \quad \text{(Red App)} \tag{14}$$

$$\Rightarrow_1\ ((\mathsf{next}\ 1.5), 0.35, [\,]) \quad \text{(Multiple steps)} \tag{15}$$

As expected, extracting the second value gives us $0.5 + 1 = 1.5$, embedded in a next, reflecting the fact that we needed to recurse to obtain the value.

5 Big-step semantics

We define a family big-step relations, denoted $\Downarrow$, where $t\ {}^{n}\!\Downarrow^{S}_{w} G$ can be read intuitively as: evaluating t (up to depth $n \in \mathbb{N}$) using the list of random draws S yields the generalized value G with density w (see Section 3.3 for a breakdown of symbols and other notation). As mentioned in the background section, big-step semantics break down computations into smaller steps and tell you how to combine the results. This corresponds more closely to human problem solving strategies and is therefore usually considered more intuitive than small-step semantics. Because there is no freedom in evaluation strategies, big-step semantics may be used efficiently for compilation or interpretation of programs. The rules for big-step evaluation are shown in Figure 6.

$$\frac{w = \mathrm{PDF}_{(\mu,\sigma)}(c) \quad w > 0}{\mathsf{normal}\ (\mu, \sigma)\ {}^{n}\!\Downarrow^{[c]}_{w} c} \quad \text{(Big Norm)} \qquad \frac{G \in \mathcal{GV}}{G\ {}^{n}\!\Downarrow^{[]}_{1} G} \quad \text{(Big Val)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S}_{w} \mathsf{in}\ G}{\mathsf{out}\ t\ {}^{n}\!\Downarrow^{S}_{w} G} \quad \text{(Big Out)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{box}\,[\vec{x} \leftarrow \vec{t}].\ t' \quad t'[\vec{t}/\vec{x}]\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{unbox}\ t\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big Box)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{next}\ t' \quad t'\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{prev}\ t\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big PrevE)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \lambda x.\ t' \quad s\ {}^{n}\!\Downarrow^{S_2}_{w_2} V \quad t'[V/x]\ {}^{n}\!\Downarrow^{S_3}_{w_3} G}{t\ \ s\ {}^{n}\!\Downarrow^{S_1 + S_2 + S_3}_{w_1 \cdot w_2 \cdot w_3} G} \quad \text{(Big App)} \qquad \frac{\mathsf{prev}\ t[\vec{t}/\vec{x}]\ {}^{n}\!\Downarrow^{S}_{w} G}{\mathsf{prev}\,[\vec{t} \leftarrow \vec{x}].\ t\ {}^{n}\!\Downarrow^{S}_{w} G} \quad \text{(Big Prev)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{inL}\ V \quad s[V/x]\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big MatchL)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S}_{w} \mathbf{fail}}{\mathsf{next}\ t\ {}^{n}\!\Downarrow^{S}_{w} \mathbf{fail}} \quad \text{(Big Next Fail)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{inR}\ V \quad r[V/y]\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big MatchR)} \qquad \frac{t[\mathsf{fix}\ f.\ t/f]\ {}^{n}\!\Downarrow^{S}_{w} G}{\mathsf{fix}\ f.\ t\ {}^{n}\!\Downarrow^{S}_{w} G} \quad \text{(Big Fix)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} c_1 \quad s\ {}^{n}\!\Downarrow^{S_2}_{w_2} c_2 \quad c_2 \neq 0}{t/s\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} (c_1/c_2)} \quad \text{(Big Div)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathtt{tt} \quad s\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big IteT)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} (t_1, t_2) \quad t_1\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{fst}\ t\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big Fst)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathtt{ff} \quad r\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big IteF)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} (t_1, t_2) \quad t_2\ {}^{n}\!\Downarrow^{S_2}_{w_2} G}{\mathsf{snd}\ t\ {}^{n}\!\Downarrow^{S_1 + S_2}_{w_1 \cdot w_2} G} \quad \text{(Big Snd)} \qquad \frac{\mathrm{PDF}_{(\mu,\sigma)}(c) = 0}{\mathsf{normal}\ (\mu, \sigma)\ {}^{n}\!\Downarrow^{[c]}_{0} \mathbf{fail}} \quad \text{(Big Norm Fail)}$$

$$\frac{t_1\ {}^{n}\!\Downarrow^{S_1}_{w_1} \beta_1 \ \ldots\ t_{|\gamma|}\ {}^{n}\!\Downarrow^{S_{|\gamma|}}_{w_{|\gamma|}} \beta_{|\gamma|}}{\gamma(t_1, \ldots, t_{|\gamma|})\ {}^{n}\!\Downarrow^{S_1 + \ldots + S_{|\gamma|}}_{w_1 \cdot \ldots \cdot w_{|\gamma|}} \sigma_\gamma(\overline{\beta_1}, \ldots, \overline{\beta_{|\gamma|}})} \quad \text{(Big Sig)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S}_{w} V}{\mathsf{next}\ t\ {}^{n+1}\!\Downarrow^{S}_{w} \mathsf{next}\ V} \quad \text{(Big Next)}$$

$$\frac{t\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{next}\ t' \quad s\ {}^{n}\!\Downarrow^{S_2}_{w_2} \mathsf{next}\ s' \quad \mathsf{next}\ (t'\ \ s')\ {}^{n}\!\Downarrow^{S_3}_{w_3} G}{t \circledast s\ {}^{n}\!\Downarrow^{S_1 + S_2 + S_3}_{w_1 \cdot w_2 \cdot w_3} G} \quad \text{(Big Dapp)} \qquad \frac{t\ {}^{n}\!\Downarrow^{S}_{w} \mathbf{fail}}{C[t]\ {}^{n}\!\Downarrow^{S}_{w} \mathbf{fail}} \quad \text{(Big CFail)}$$

Figure 6: The rules for big-step semantics.

Note that the top of the inference rules generally ask that sub-calculations evaluate to values. The bottom of the rules then state what to do with these values when they are encountered in some setting. Evaluation therefore no longer takes place within contexts as defined in Section 4. We still use contexts to propagate **fail** terms up the evaluation tree in the cases where this is not done automatically. The random draws lists are no longer used up incrementally. Instead, the lists are split between the sub-calculations, which each use up their part. Similarly, the densities from sub-calculations are combined through multiplication in the total calculation.

The interpreter based on these semantics simply evaluates each sub-calculation, taking the draws it needs from the draws list and passing the remainder of the list to the next sub-calculation. This strategy is efficient and easy to implement.

6 Big-step and small-step equivalence

To prove that the small-step and big-step semantics we have given are equivalent, we show that they give the same program traces and densities when given identical random draw lists. To do so, we first show invariance of the big-step semantics on the small-step semantics.

Lemma 13 (Invariance)**:**
If $(t, w^*, S) \to_n (t', w^* \cdot w, [\,])$ *for some* $w^* > 0$ *and* $t' \Downarrow^{S'}_{w'} G$*, then* $t \Downarrow^{S +\!\!+ S'}_{w \cdot w'} G$

The proof for this will rest on the inversion of the big step rules from Figure 6. Whenever there is no unique inversion, all possible inversions are dealt with seperately. The structure of this proof, and some of the Lemma's presented before, are inspired by the equivalence proof for a probabilistic lambda calclus by Borgström et al [6]. We only give a few interesting cases here, but the entire proof can be found in Appendix A.

We implicitly make use of the (Big Val) rule, under which values relate to themselves. This step is writtin out in Case 13.2, but omitted in later cases (in the appendix). Note that w^* may not be 0, as this would cause solutions for w to the formula $w^* = w^* \cdot w$ to no longer be unique, which is needed to proof cases like Case 13.2.

Proof. If $t = C[\textbf{fail}]$ for some $C \neq [\cdot]$ or $t = \mathsf{next}\ \textbf{fail}$, then by (Big CFail) or (Big Next Fail), we get $t' = \textbf{fail}$ and thus by (Big Val) we get $G = \textbf{fail}$. Then by (Big CFail) or (Big Next Fail) $t \Downarrow^{S +\!\!+ [\,]}_{w \cdot 1} G$. Otherwise, by induction on the structure of C:

Base case: t is a redex. So case by analysis (analogous cases are grouped together) on t:

Case 13.1 ($\mathsf{next}\ t,\ \gamma(t_1, \ldots, t_{|\gamma|})$)
Here t reduces to some G in one reduction step. Thus by (Big Val) $t' \Downarrow^{S'}_{w'} G$. The desired result then follows from the big-step rule that corresponds with the redex.

Case 13.2 ($\mathsf{fst}\ (t, s)$, $\mathsf{snd}\ (t, s)$, $\mathsf{out}\ (\mathsf{in}\ t)$, $\mathsf{prev}\ (\mathsf{next}\ t)$, $\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ t\ \mathsf{else}\ s$, $\mathsf{if}\ \mathtt{ff}\ \mathsf{then}\ t\ \mathsf{else}\ s$)
By (Red Fst), $(\mathsf{fst}\ (t, s), w^*, [\,]) \to_n (t, w^*, [\,])$ and thus $w = 1$ and $S = [\,]$. By assumption, $t \Downarrow^{S'}_{w'} G$. By (Big Val), $(t, s) \Downarrow^{[\,]}_1 (t, s)$. Then by (Big Fst), the desired result is obtained.

Case 13.3 ($\mathsf{normal}\ (\mu, \sigma,)$)
Here, there are two possibilities:

- If $(t, w^*, S) \to_n (t', w^* \cdot w, [\,])$ was derived with (Small Norm), then $S = [c]$ and $w = \mathrm{PDF}_{(\mu,\sigma)}(c) > 0$. Then, by (Big Norm), $\mathsf{normal}\ (\mu, \sigma)\ {}^{n}\!\Downarrow_{w}^{S} c$
- If $(t, w^*, S) \to_n (t', w^* \cdot w, [\,])$ was derived with (Small Fail), then $S = [c]$ and $w = \mathrm{PDF}_{(\mu,\sigma)}(c) = 0$. Then by (Big Norm Fail), $\mathsf{normal}\ (\mu, \sigma)\ {}^{n}\!\Downarrow_{w}^{S} \mathbf{fail}$

...

Induction step: Assume Lemma 13 holds for some $s = C[R]$ with $R \neq \mathbf{fail}$, then for some $C' \neq [\cdot]$, we take $t = C'[C[R]]$. Now by case analysis on C':

Case 13.4 ($\mathsf{out}\ [\cdot]$)

$(\mathsf{out}\ C[R], w^*, S) \to_n (\mathsf{out}\ C[R_1], w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$\mathsf{out}\ C[R_1]\ {}^{n}\!\Downarrow_{w'}^{S'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)
$C[R_1]\ {}^{n}\!\Downarrow_{w_1}^{S_1} \mathsf{in}\ G$	(By 2, inversion of Big Out)	(4)
$C[R]\ {}^{n}\!\Downarrow_{w \cdot w_1}^{S +\!\!+ S_1} \mathsf{in}\ G$	(By induction hypothesis)	(5)
$\mathsf{fst}\ C[R]\ {}^{n}\!\Downarrow_{w \cdot w_1}^{S +\!\!+ S_1} G$	(By Big Out)	(6)

Where $S' = S_1$ and $w' = w_1$, giving the desired result.

Case 13.5 ($\mathsf{if}\ [\cdot]\ \mathsf{then}\ t\ \mathsf{else}\ s_0$)

$(\mathsf{if}\ C[R]\ \mathsf{then}\ t\ \mathsf{else}\ s_0, w^*, S) \to_n$ $(\mathsf{if}\ C[R_1]\ \mathsf{then}\ t\ \mathsf{else}\ s_0, w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$(\mathsf{if}\ C[R_1]\ \mathsf{then}\ t_0\ \mathsf{else}\ s_0)\ {}^{n}\!\Downarrow_{w'}^{S'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)

- If $(\mathsf{if}\ C[R_1]\ \mathsf{then}\ t_0\ \mathsf{else}\ s_0)\ {}^{n}\!\Downarrow_{w'}^{S'} G$ was derived using (Big IteT), then

$C[R_1]\ {}^{n}\!\Downarrow_{w_1}^{S_1} \mathtt{tt}, \quad t_0\ {}^{n}\!\Downarrow_{w_2}^{S_2} G$	(By assumption)	(1)
$C[R]\ {}^{n}\!\Downarrow_{w \cdot w_1}^{S +\!\!+ S_1} \mathtt{tt}$	(By induction hypothesis)	(2)
$C[R]\ {}^{n}\!\Downarrow_{w \cdot w_1 \cdot w_2}^{S +\!\!+ S_1 +\!\!+ S_2} G$	(By Big IteT)	(3)

  Where $S' = S_1 +\!\!+ S_2$ and $w' = w_1 \cdot w_2$, giving the desired result.

- If $(\mathsf{if}\ C[R_1]\ \mathsf{then}\ t_0\ \mathsf{else}\ s_0)\ {}^{n}\!\Downarrow_{w'}^{S'} G$ was derived using (Big IteF), the reasoning is analogous.

...

□

We can now proof the equivalence of the big-step and small-step semantics. We once again only proof a few interesting cases here. The full proof can be found in Appendix B.

Theorem 1. *$t \Downarrow_w^S G$ if and only if $(t, 1, S) \Rightarrow_n (G, w, [\,])$*

Proof. The left-to-right implication is proven using induction on the derivation of $t \Downarrow_w^S G$ (analogous cases are grouped together). Note that the induction hypothesis here assumes equivalence of the tops of the big-step rules.

Case 1.1 (Big CFail)
By induction hypothesis: $(t, 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$. Then by Red CFail and Small Red:
$(C[t], 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$

Case 1.2 (Big Norm Fail)
By assumption, $\mathrm{PDF}_{(\mu,\sigma)}(c) = 0$. Then by Small Fail: $(\mathsf{normal}\ (\mu, \sigma), 1, [c]) \Rightarrow_n (\mathbf{fail}, 0, [\,])$

Case 1.3 (Big Out)

$(t, 1, S) \Rightarrow_n (\mathsf{in}\ G, w, [\,])$	(By induction hypothesis)	(1)
Using $C = \mathsf{out}\ [\cdot] :\ (\mathsf{out}\ t, 1, S) \Rightarrow_n (\mathsf{out}\ (\mathsf{in}\ G), w, [\,])$	(By Lemma 10)	(2)
Using $C = [\cdot]$ and Red Out $(\mathsf{out}\ (\mathsf{in}\ G), w, [\,]) \rightarrow_n (G, w, [\,])$	(By Small Red)	(3)
$(\mathsf{out}\ t, 1, S) \Rightarrow_n (G, w, [\,])$	(2, 3)	(4)

Case 1.4 (Big App, Big Dapp)

$(t, 1, S_1) \Rightarrow_n (\lambda x.\ t', w_1, [\,])$, $(s, 1, S_2) \Rightarrow_n (V, w_2, [\,])$, $(t'[V/x], 1, S_3) \Rightarrow_n (G, w_3, [\,])$	(By induction hypothesis)	(1)
Using $C = [\cdot]\ s$ $(t\ s, 1, S_1) \Rightarrow_n ((\lambda x.\ t')\ s, w_1, [\,])$	(By Lemma 10)	(2)
$(t\ s, 1, S_1 +\!\!+ S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ s, w_1, S_2 +\!\!+ S_3)$	(By Lemma 11)	(3)
Using $C = (\lambda x.\ t')\ [\cdot]$ $((\lambda x.\ t')\ s, 1, S_2) \Rightarrow_n ((\lambda x.\ t')\ V, w_2, [\,])$	(By 1, Lemma 10)	(4)
$((\lambda x.\ t')\ s, 1, S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ V, w_2, S_3)$	(By Lemma 11)	(5)
$((\lambda x.\ t')\ s, w_1 \cdot 1, S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ V, w_1 \cdot w_2, S_3)$	(By Lemma 12)	(6)
$(t\ s, 1, S_1 +\!\!+ S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ V, w_1 \cdot w_2, S_3)$	(By 3, 6)	(7)
Using $C = [\cdot]$ and Red App $((\lambda x.\ t')\ V, w_1 \cdot w_2, S_3) \rightarrow_n (t'[V/x], w_1 \cdot w_2, S_3)$	(By Small Red)	(8)
$(t'[V/x], w_1 \cdot w_2 \cdot 1, S_3) \Rightarrow_n (G, w_1 \cdot w_2 \cdot w_3, [\,])$	(By 1, Lemma 12)	(9)
$(t\ s, 1, S_1 +\!\!+ S_2 +\!\!+ s_3) \Rightarrow_n (G, w_1 \cdot w_2 \cdot w_3, [\,])$	(By 7, 8, 9)	(10)

...

The right-to-left implication is proved using induction on the small-step derivation, making use of Lemma 13. We start by proving a more general version of the right-to-left implication:

$$\text{For all } w^* > 0, \text{ if } (t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,]) \text{ then } t \,{}^n\!\Downarrow_w^S G$$

The desired result is then obtained by simply taking $w^* = 1$. By induction on the derivation of $(t, w^*, S) \Rightarrow_n (G, w \cdot w^*, [\,])$:

Base case: If $(t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,])$ by reflexivity, then $t = G$, $w = 1$ and $S = [\,]$. By (Big Val), we get $t \,{}^n\!\Downarrow_1^{[]} G$, which is the desired result.

Induction step: Assume that $(t, w^*, S) \rightarrow_n (t', w', S') \Rightarrow_n (G, w \cdot w^*, [\,])$ for some (t', w', S'). Then there are w_1 and w_2, such that $(t, w^*, S) \rightarrow_n (t', w^* \cdot w_1, S') \Rightarrow_n (G, w^* \cdot w_1 \cdot w_2, [\,])$, where $w' = w^* \cdot w_1$ for some w_1 and $w = w_1 \cdot w_2$ for some w_2. The induction hypothesis then gives $t' \,{}^n\!\Downarrow_{w_2}^{S'} G$.

$$t' \,{}^n\!\Downarrow_{w_2}^{S'} G \qquad \text{(By induction hypothesis)} \qquad (1)$$

$$(t, w^*, S'') \rightarrow_n (t', w^* \cdot w_1, [\,]), \text{ with } S = S'' +\!\!+ S' \qquad \text{(By Lemma 1)} \qquad (2)$$

$$t \,{}^n\!\Downarrow_{w_1 \cdot w_2}^{S'' +\!\!+ S'} G \qquad \text{(By Lemma 13)} \qquad (3)$$

Thus, by induction on the multi-step derivation, noting that $S'' +\!\!+ S' = S$ and $w_1 \cdot w_2 = w$: If $(t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,])$ then $t \,{}^n\!\Downarrow_w^S G$.
Finally, as the implication holds both ways, Theorem 1 holds. □

7 Related work

As mentioned in the introduction, the language we analyse is closely related to the one described by Birkedal et al. [5], Henning Basold [4] and others. There are two main problems in trying to give semantics for GHOPFL: its probabilistic nature and the productivity guarantees.

A useful insight for writing down the semantics of a probabilistic language was provided by Dexter Kozen [15]. He proposed simply giving the semantics, parametrised by a list of random draws that occur during program execution. Using this strategy, one no longer assigns a meaning to program, but one assigns a meaning and a density to a program and a draws list. The complete semantics of a program may then be thought of as a probability distribution over all possible draws lists. This simplifies writing down the semantics, at the cost of less straight-forward interpretation of programs. Borgström et al. give the semantics for a probabilistic lambda calculus in [6] using Kozen's insight. We adopt much of the notation and use insights from this work.

The notions regarding productivity in this thesis are centred around Nakano's later modality [18]. It forms a strong basis for productive co-programming, but is limited in its uses as it is very restrictive. Many works trying to alleviate these restrictions followed. Atkey and McBride proposed the idea of clock variables, allowing for finite observations on infinite data [2]. A second modality, granting some of the powers of clock variables was introduced by Clouston et al. [8]. This modality is called

the *constant* modality and is featured in this thesis. Adrien Guatto even proposed a more general modality for time displacement [10], of which the later and constant modalities are special cases.

8 Example programs

To showcase the utility of a language like GHOPFL, we give some examples of programs that take advantage of the coinductive and probabilistic nature of GHOPFL and analyse their big-step execution. All the programs listed here, have an analogue in the code base [24]. We start by explaining some additions to the language, present in the intepreter, that aid the programmer in writing more complicated programs.

Prefix synonyms As we intend to do higher order programming with operations like "+" and "/", we define prefix synonyms for these operations. For example:

$$\texttt{add} \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x + y \qquad\qquad \texttt{divide} \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x/y$$

Random Our language, as described, only supports the sampling of normal distributions. When implementing algorithms however, a random function that returns a uniformly random value in $[0, 1]$ is often useful. We will use such a function in the example programs under the name `rand`. Uniformity of `rand` could be achieved by realising that $\mathrm{CDF}(X)$ of a normally distributed variable X, is uniform in $[0, 1]$. Alternatively, one could simply imagine adding a second sampling primitive. The interpreter takes the second approach.

Integers To keep the semantics compact, we have thus far only considered real numbers. The interpreter also supports integers, which will mostly be used as indexes in the example programs.

Lists As lists are very useful when implementing many common algorithms, we have included them in the implementation. They are constructed using the regular $[\ldots]$ syntax and many common operations are defined on them.

Delayed Results Often times, when evaluating guarded programs, the results will be delayed by some arbitrary amount of steps. An example is the indexation of guarded streams (see Section 3.4.1). Recall that the tail of such a stream is of type $(\blacktriangleright\, \mathbf{gStr}\, A)$. As the second element is in the tail, it must be of type $\blacktriangleright A$. In fact, getting deeper and deeper values from a guarded stream looks like:

$$\begin{aligned}
\texttt{g2nd} &\stackrel{\text{def}}{=} \lambda s.\ \texttt{next}\ \texttt{glHead} \circledast \texttt{glTail}\ s &&: \quad \mathbf{gStr}\, A \to \blacktriangleright A \\
\texttt{g3rd} &\stackrel{\text{def}}{=} \lambda s.\ \texttt{next}\ \texttt{g2nd} \circledast \texttt{glTail}\ s &&: \quad \mathbf{gStr}\, A \to \blacktriangleright \blacktriangleright A \\
\texttt{g4th} &\stackrel{\text{def}}{=} \lambda s.\ \texttt{next}\ \texttt{g3rd} \circledast \texttt{glTail}\ s &&: \quad \mathbf{gStr}\, A \to \blacktriangleright \blacktriangleright \blacktriangleright A
\end{aligned}$$

A guarded stream indexation function must be able to return all these types. We define the type:

$$\mathbf{Del}\, A \stackrel{\text{def}}{=} \mu X.\ A + \blacktriangleright X$$

It represents data that is delayed by some arbitrary (possibly infinite) amount. We can now write our indexation function:

$$\texttt{gIdx} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda n.\ \lambda s.\ \mathsf{if}\ (n = 0)\ \mathsf{then\ in\ inL}\ (\texttt{gHead}\ \ s)\ \mathsf{else\ in\ inR}\ (f \circledast \mathsf{next}\ (n-1) \circledast (\texttt{gTail}\ \ s))$$

It has type $(\mathbf{int} \to \mathbf{gStr}\, A \to \mathbf{Del}\, A)$ and can thus return the element at any index, embedded in the appropriate amount of delay. As many computations will give such delayed results, we will now define some helpful notions around them. We can construct values of type $\mathbf{Del}\, A$ using:

$$\texttt{now} \stackrel{\text{def}}{=} \lambda x.\ \mathsf{in\ inL}\ x : A \to \mathbf{Del}\, A \qquad \texttt{later} \stackrel{\text{def}}{=} \lambda x.\ \mathsf{in\ inR}\ x : \blacktriangleright\, \mathbf{Del}\, A \to \mathbf{Del}\, A$$

Assume we wanted to use a result of type $\mathbf{Del}\, A$ in another calculation. We can do this by lifting functions of type $A \to B$ using:

$$\begin{aligned} &\texttt{dLift} \stackrel{\text{def}}{=} \lambda f.\ \mathsf{fix}\, r.\ \lambda x.\ \mathsf{match}\ (\mathsf{out}\ x) \\ &\qquad \{\ \mathsf{inL}\ x \mapsto \texttt{now}\ \ (f\ \ x) \\ &\qquad ;\ \mathsf{inR}\ y \mapsto \texttt{later}\ \ (r \circledast y)\ \} \end{aligned} \quad : \quad \begin{aligned} &(A \to B) \\ &\to (\mathbf{Del}\, A \to \mathbf{Del}\, B) \end{aligned}$$

It is important to note that the amount of delay is not changed in the result. We can also lift functions of higher arity. To do so, we first define a function that acts on a regular value a and a delayed value d.

$$\begin{aligned} &\texttt{dLift2}' \stackrel{\text{def}}{=} \lambda f.\ \mathsf{fix}\, r.\ \lambda a.\ \lambda d.\ \mathsf{match}\ (\mathsf{out}\ d) \\ &\qquad \{\ \mathsf{inL}\ x \mapsto \texttt{now}\ \ (f\ \ a\ \ x) \\ &\qquad ;\ \mathsf{inR}\ y \mapsto \texttt{later}\ \ (r \circledast \mathsf{next}\ a \circledast y)\ \} \end{aligned} \quad : \quad \begin{aligned} &(A \to B \to C) \\ &\to A \to \mathbf{Del}\, B \to \mathbf{Del}\, C \end{aligned}$$

Then, using $\texttt{dLift2}'$ and $(\texttt{flip} \stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ \lambda y.\ f\ \ y\ \ x)$, we can write:

$$\begin{aligned} &\texttt{dLift2} \stackrel{\text{def}}{=} \lambda f.\ \mathsf{fix}\, r.\ \lambda x_1.\ \lambda x_2.\ \mathsf{match}\ (\mathsf{out}\ x_1) \\ &\qquad \{\ \mathsf{inL}\ a_1 \mapsto \texttt{dLift2}'\ \ f\ \ a_1\ \ x_2 \\ &\qquad ;\ \mathsf{inR}\ d_1 \mapsto \mathsf{match}\ (\mathsf{out}\ x_2) \\ &\qquad\qquad \{\ \mathsf{inL}\ a_2 \mapsto \texttt{dLift2}'\ \ (\texttt{flip}\ \ f)\ \ a_2\ \ x_1 \\ &\qquad\qquad ;\ \mathsf{inR}\ d_2 \mapsto \texttt{later}\ \ (r \circledast d_1 \circledast d_2)\ \}\ \} \end{aligned} \quad : \quad \begin{aligned} &(A \to B \to C) \\ &\to \mathbf{Del}\, A \to \mathbf{Del}\, B \to \mathbf{Del}\, C \end{aligned}$$

It should be noted that the amount of delay in the resulting value is equal to the amount of delay in the most delayed input: We can only perform our calculation once all the data involved has become available. Care is taken in the implementation of `dLift2` to not introduce more delay than necessary. We will be using lifted functions like this in some of the examples.

8.1 Random walk

A random walk is a random process where random steps are taken across some mathematical space, like the number line. We will implement a (normally distributed) random walk across $\mathbb{R}$. Given some point $x \in \mathbb{R}$, the next point will be a random sample from a normal distribution with mean x and some fixed variance s. As random walks are infinite processes, we choose to describe them using a guarded stream (see Section 3.4.1):

$$\texttt{RW} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda x.\ \mathsf{in}\ (\ x, f \circledast \mathsf{next}\ (\mathsf{normal}\ (\ x, s\ ))\ )$$

At every step, the guarded stream is built up using the current value and a recursive call with argument $\mathsf{normal}\ (x, s)$. To instantiate the stream, we must pass the stream an initial value i and provide an s in the environment. We first define a function that takes two guarded streams and returns the element-wise product:

$$\texttt{gProduct} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda s_1.\ \lambda s_2.\ \mathsf{in}\ (\,(\,\texttt{gHead}\ s_1, \texttt{gHead}\ s_2\,), f \circledast (\texttt{gTail}\ s_1) \circledast (\texttt{gTail}\ s_2)\,)$$

Using `gProduct`, we can make random walks over $\mathbb{R}^n$:

$$\texttt{RW}_n \stackrel{\text{def}}{=} \texttt{gProduct}\ (\texttt{RW}\ i_1)\ \texttt{gProduct}\ (\texttt{RW}\ i_2)\ \ldots\ \texttt{gProduct}\ (\texttt{RW}\ i_{n-1})\ (\texttt{RW}\ i_n)$$

Random walks have many applications. They have been used as a simplified model for Brownian motion [13] and as a method of estimating the size of the world wide web [3]. In both these cases, analysis or other calculations are performed on the random walks. This means these programs could benefit from the advantages of coinduction, as described in the introduction.

8.2 Geometric distribution

A typical example within probabilistic programming is the geometric distribution. This distribution concerns a series of independent events, which can have two disjoint outcomes. One of the outcomes is usually referred to as a "success" and has probability p of occurring. The other is called a "failure" and occurs with probability $1 - p$. The geometric distribution is defined as the probability distribution of the number of successes before a failure occurs. A common concrete example is coin toss with a biased coin that has probability p of landing heads up. We consider the coin landing tails up a success and heads up a failure. Assume we wanted to know how many flips, on average, it takes before the first heads occurs in a sequence of coin tosses. We might try and do mathematical analysis, but we might also make a model in our probabilistic language. Firstly we define a function that takes a single sample from the geometric distribution (parameterized by p). Its return type is $(\mathbf{Del}\,(\mathbf{int}))$, reflecting the fact that any number of heads may occur before the first tails occurs.

$$\texttt{geometric} \stackrel{\text{def}}{=} \lambda p.\ \mathsf{fix}\, f.\ \mathsf{if}\ (\texttt{rand} < p)\ \mathsf{then}\ (\texttt{now}\ 0)\ \mathsf{else}\ \texttt{dLift}\ (\texttt{add}\ 1)\ (\texttt{later}\ f))$$

This recursive function takes p and compares it against samples from a uniform distribution over [0,1] until it's found a sample smaller than p. Let us define a mean approximation process. We start by creating a stream of samples.

$$\texttt{sampleStream} \stackrel{\text{def}}{=} \lambda p.\ \mathsf{fix}\, f.\ \mathsf{in}\ (\,\texttt{geometric}\ p, f\,)$$

We now define a function that calculates a new mean, given one new data point. Here μ is the current mean estimate, c is the amount of samples considered in this mean estimate and n is the new data point. Note that we have to lift our arithmetic operators, as `geometric` recurses and must thus give delayed results.

$$\begin{aligned}\texttt{meanStep} \stackrel{\text{def}}{=} \lambda\mu.\ \lambda c.\ \lambda c.\ &\texttt{dLift2 divide}\\ &(\texttt{dLift2 add}\ (\texttt{dLift2 multiply}\ \mu\ c)\ n)\\ &(\texttt{dLift}\ (\texttt{add}\ 1)\ c)\end{aligned}$$

This function is very sensitive to rounding errors and should thus be implemented using arbitrary-precision number types. As we are reasoning theoretically here, we will discard this concern. We

can now create a causal stream function, whose n-th output is the mean of the first n numbers of the input stream. Once again, we keep track of the mean so far (μ) and the amount of samples (c). Here, s is the input stream.

$$\begin{aligned}\texttt{streamMean} \stackrel{\text{def}}{=} (\mathsf{fix}\ f.\ \lambda\mu.\ \lambda c.\ \lambda s.\ \mathsf{in}\ (\ \mu,\\ f \circledast \mathsf{next}\ (\texttt{meanStep}\ \mu\ c\ (\texttt{gHead}\ s))\\ \circledast \mathsf{next}\ (\texttt{dLift}\ (\texttt{add}\ 1)\ c)\\ \circledast (\texttt{gTail}\ s)\ )\\ )\ (\texttt{now}\ 0.0)\ (\texttt{now}\ 0)\end{aligned}$$

The starting points for the mean and index are already provided in the outer-most applications. We can now extract iteration k from the mean approximation process using guarded stream indexation:

$$\texttt{gIdx}\ k\ (\texttt{streamMean}\ (\texttt{sampleStream}\ p))$$

The power of such a mean approximation program becomes more apparent when dealing with more complicated models. The code base contains a more complicated model that estimates the energy production of a windmill using multiple distributions, combined using programming constructs. The (not very realistic) situation is as follows:

A farmer wants to invest in a windmill on his property. The windmill company gives the following statistics regarding the energy production of the windmill:

1. Each day, the windmill has a 1% chance of having a major failure, which will cost 20.000 kWh to fix.
2. If there is no such failure, the windmill will generate energy proportional to the average wind speed that day. The average wind speed is Rayleigh distributed and the windmill generates 5000 kWh per speed unit.
3. This energy production may be reduced by minor failures, which each decrease the energy production by a factor of 0.95. The amount of minor failures that occur on a day are Poisson distributed.

The farmer would now like to calculate the mean energy produced per day. He is not well versed in statistical modelling, but he is a decent programmer. As such, he writes a coinductive mean approximation process like the one above.

As the program representing this process is quite large, we will not break it down here, but it may be examined in the code base. All the distributions mentioned may be generated using `rand` or the native normal term. In the case of this process, the way in which the various distributions combine, makes it much harder to reason about them mathematically, showing the utility of the programmatic approach.

8.3 Simulated annealing

Another interesting application of the co-inductive features of GHOPFL is optimization algorithms. Optimisation algorithms can be thought of as infinite streams of (hopefully) better and better

guesses. We define such a coinductive stream (see Section 3.4.3) for the simulated annealing algorithm and query an index of the stream to find the optimization result at a given iteration. The algorithm is iterative and works as follows:

Algorithm 1 Simulated annealing algorithm

```
1: T ← init_T,   s ← init_s
2: loop
3:     s' ← mutate s
4:     if f(s') < f(s) then
5:         s ← s'
6:     else
7:         if rand < exp(−((f(s') − f(s))/T)) then
8:             s ← s'
9:         end if
10:    end if
11:    T ← T ∗ coolfactor
12: end loop
```

Here, T is the temperature of the annealing process, s is the current state, f is the evaluation function and `rand` samples a uniform random value in $[0, 1]$. The algorithm starts with some initial temperature $\texttt{init}_\texttt{T}$ and initial state $\texttt{init}_\texttt{s}$. Each iteration, the state is mutated in a small way. This mutated state is then non-deterministically selected over the current state. At the end of each iteration, the temperature is brought down by the `coolfactor`, which should be less than 1. Note that a cooldown factor is only one of many ways in which the temperature may be brought down at the end of each iteration. Note that there is no stop-condition for the main loop. Normally, the stop-condition is enforced arbitrarily, either as an iteration budget, or as a sufficiency measure. As we are working coinductively, we write the algorithm without a stop-condition.

We will apply this algorithm to the Max-Cut problem for the graph in Figure 7. The Max-Cut

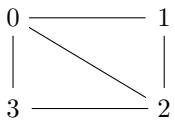
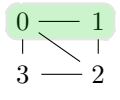


Figure 7: The graph to optmize for.

problem is the problem of finding the partition of a graph, for which a boundary line between the two partitions cuts the most edges possible. Examples of partitions and the value the evaluation function assigns them are given in Table 1.

Partition	f
	3

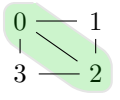
Partition	f
	4

Table 1: Example partitions.

To implement the algorithm in GHOPFL, we will store the graph as an adjacency list:

$$\texttt{graph} \stackrel{\text{def}}{=} [(0,1),(0,2),(0,3),(1,2),(2,3)]$$

We will represent solution vectors as a list of booleans, one for each node, where `tt` and `ff` represent the two partitions a node can be in. We start with the state:

$$\texttt{init} \stackrel{\text{def}}{=} [\texttt{tt},\texttt{tt},\texttt{ff},\texttt{ff}]$$

To evaluate a state, we first define a function that takes a state and an edge, and returns 1 when that edge is cut, 0 otherwise. Here we use the function $\texttt{xor} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ (p \vee q) \wedge \neg(p \wedge q)$

$$\texttt{cuts} \stackrel{\text{def}}{=} \lambda s.\ \lambda edge.\ \mathsf{if}\ (\texttt{xor}\ \ s[\mathsf{fst}\ edge]\ \ s[\mathsf{snd}\ edge])\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$

Equipped with this function, we can write a fold on the adjacency list that returns how many edges are cut by a given state vector:

$$\texttt{cutCount} \stackrel{\text{def}}{=} \lambda s.\ \texttt{foldl add 0 (map (cuts}\ s)\ \texttt{graph)}$$

As simulated annealing optimizes for lower energy solutions, we invert the result of `cutCount` to obtain our evaluation function: $(\texttt{f} \stackrel{\text{def}}{=} \lambda s.\ 1.0/(\texttt{cutcount}\ s))$. To perform mutation, we first create a function that flips the bit of the state at index idx:

$$\texttt{flipBit} \stackrel{\text{def}}{=} \lambda idx.\ \lambda s.\ (\texttt{take}\ \ idx\ \ s)\ +\!\!+\ [\,\neg s[idx]\,]\ +\!\!+\ (\texttt{drop}\ \ (idx+1)\ \ s)$$

To `flipBit` at a random index, we introduce a function that uniformly randomly returns an integer in $\{0,1,2,3\}$, called `randIdx4` . This function might easily be implemented by dividing up $[0,1]$ into 4 indexed, disjoint intervals of size $\frac{1}{4}$ and returning the index of the interval in which a `rand` sample falls.

$$\texttt{mutate} \stackrel{\text{def}}{=} \texttt{flipBit randIdx4}$$

Selection is relatively straight forward. A solution is compared against its mutation and non-deterministically selected using `rand`.

$$\texttt{select} \stackrel{\text{def}}{=} \lambda s.\ \lambda s'.\ \lambda T.\ \mathsf{if}\ (\texttt{f}\ \ s' < \texttt{f}\ \ s)\ \mathsf{then}\ s'\ \mathsf{else}\ (\mathsf{if}\ (\texttt{rand} < (\texttt{p}\ \ s\ \ s'\ \ T))\ \mathsf{then}\ s'\ \mathsf{else}\ s)$$

Here s is the current solution and s' is the mutated s. Furthermore, `p` calculates the probability in the if-statement on line 7 of the algorithm pseudocode.

Finally, we define a function `getT`, which calculates the desired temperature for a given iteration. We define it to simply return $\texttt{initT}/(k+1)$, with k the current iteration and `initT` some initial temperature, which we will take to be 1.0.

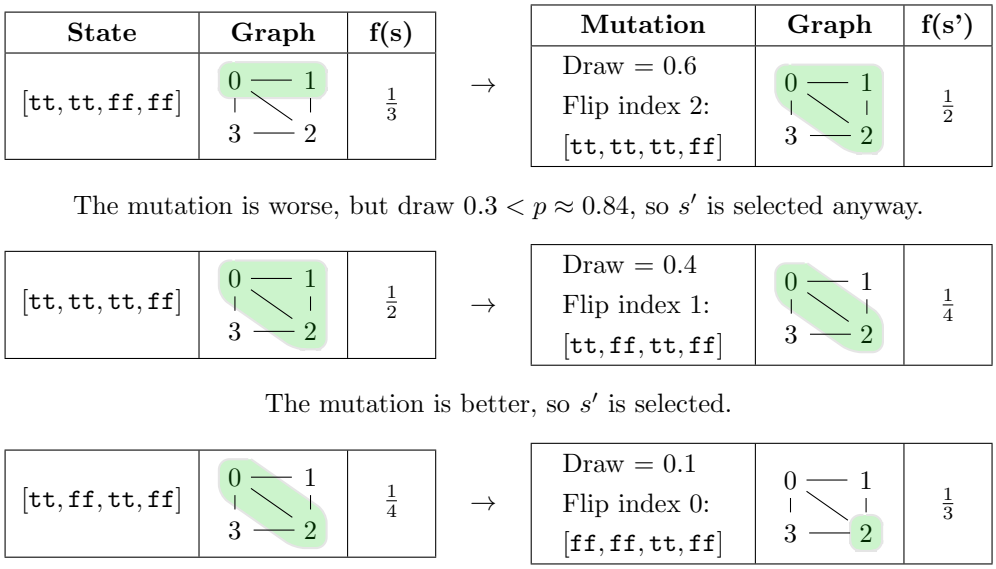


Figure 8: A simulated annealing example run

Now we have everything we need to define a stream of optimization iterations:

$$\texttt{oStr} \stackrel{\text{def}}{=} \mathsf{fix}\, f.\ \lambda k.\ \lambda s.\ \mathsf{in}\ (\ s, f \circledast \mathsf{next}\ (k+1) \circledast \mathsf{next}\ (\texttt{select}\ s\ (\texttt{mutate}\ s)\ (\texttt{getT}\ k))\ )$$

Here, k represents the current iteration and s represents current state. We can now pass this description of the optimisation process around with all the advantages of coinductive programming. For now we simply extract the third iteration by giving `oStr` initial values for k and s, boxing the stream to make it coinductive and applying `bHead` and `bTail`:

$$\texttt{get3rd} \stackrel{\text{def}}{=} \lambda s.\ \texttt{bHead}\ (\texttt{bTail}\ (\texttt{bTail}\ s))$$

$$\texttt{result} \stackrel{\text{def}}{=} \texttt{get3rd}\ (\mathsf{box}\, \iota.\ \texttt{oStr}\ 0\ \texttt{init})$$

The code base contains an example for this algorithm, the execution of which is shown in Figure 8. This example assumes the random draws $[0.6, 0.3, 0.4, 0.1, 0.9]$.

9 Conclusion

This thesis presented an operational semantics basis for a higher order, probabilistic, productively coinductive programming language, called GHOPFL. We gave the syntax for this language and motivated its terms. We presented a small-step and a big-step semantics. We proved equivalence between the two semantics, meaning one may choose either as a starting point when reasoning about the language. We gave examples of programs that can benefit from the features of GHOPFL and

we have provided an interpreter of the language in which these examples may be run and examined further.

9.1 Future work

As mentioned in Section 7, there are many proposed strategies for relaxing the constraints around guarded recursion. Only one of them is analysed in this thesis, but incorporating others could improve the computational power of the language or simplify the semantics.

The approach chosen to deal with the probabilistic part of GHOPFL, namely a sampling based semantics, is not the only one available. Borgström et al. describe a distribution based semantics [6], rooted in measure theory, that may possibly be extended to fit a guarded recursive language like GHOPFL.

Henning Basold's denotational semantics [4] for a guarded recursive language, which inspired the creation of GHOPFL, does not incorporate the constant modality. Working out a denotational semantics that does include the constant modality would round out the semantics for GHOPFL.

Finally, there are some desirable functionalities that the Haskell interpreter does not yet have. An example is a type checker. As it is currently, evaluation simply fails when the wrong type of argument is encountered during evaluation. This means there are lots of explicit error handling steps in the code. Furthermore, writing large programs becomes more managable when the programmer is able to type-check their programs.

References

[1] Andreas Abel and Andrea Vezzosi. A formalized proof of strong normalization for guarded recursive types. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 140–158, Cham, 2014. Springer International Publishing.

[2] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. *SIGPLAN Not.*, 48(9):197–208, September 2013.

[3] Ziv Bar-Yossef and Maxim Gurevich. Random sampling from a search engine's index. *J. ACM*, 55(5), November 2008.

[4] Henning Basold. Coinduction in Flow: The Later Modality in Fibrations. In Markus Roggenbach and Ana Sokolova, editors, *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, volume 139 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), Oct 2012.

[6] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. *CoRR*, abs/1512.08990, 2015.

[7] Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[8] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 407–421, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models, 2014.

[10] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 482–491, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Helle Hansen, Clemens Kupke, and Jan Rutten. Stream differential equations: Specification formats and solution methods. *Logical Methods in Computer Science*, 13(1):1–51, February 2017.

[12] Olav Kallenberg. *Foundations of modern probability*. Probability and its Applications. Springer-Verlag, New York, second edition, 2002.

[13] Frank B. Knight. On the random walk and brownian motion. *Transactions of the American Mathematical Society*, 103(2):218–228, 1962.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, pages 137–138. Addison-Wesley, Boston, third edition, 1997.

[15] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[16] KU Leuven, DTAI Research Group. 2020. Problog. `https://problog.readthedocs.io/en/latest/`.

[17] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[18] Hiroshi Nakano. A modality for recursion, 2000.

[19] Marco Patrignani, Eric Mark Martin, and Dominique Devriese. On the semantic expressiveness of recursive types, 2020.

[20] Yura N. Perov and Frank D. Wood. Automatic sampler discovery via probabilistic programming and approximate bayesian computation. In *AGI*, 2016.

[21] Raul Rojas. A tutorial introduction to the lambda calculus, 2015.

[22] Stan Development Team. 2019. Stan modeling language users guide and reference manual, 2.27. `https://mc-stan.org/`.

[23] David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. *arXiv preprint arXiv:1608.05263*, 2016.

[24] Rintse van de Vlasakker. GHOPFL: A Haskell big-step implementation. `https://github.com/Rintse/ghopfl`, 2021.

Appendices

A Complete invariance proof

Proof. If $t = C[\mathbf{fail}]$ for some $C \neq [\cdot]$ or $t = \mathsf{next}\ \mathbf{fail}$, then by (Big CFail) or (Big Next Fail), we get $t' = \mathbf{fail}$ and thus by (Big Val) we get $G = \mathbf{fail}$. Then by (Big CFail) or (Big Next Fail) $t \overset{n}{\Downarrow}{}^{S+[]}_{w\cdot 1} G$. Otherwise, by induction on the structure of C:

Base case: t is a redex. So case by analysis (analogous cases are grouped together) on t:

Case 13.6 ($\mathsf{next}\ t,\ \gamma(t_1, \ldots, t_{|\gamma|})$)
Here t reduces to some G in one reduction step. Thus by (Big Val) $t' \overset{n}{\Downarrow}{}^{S'}_{w'} G$. The desired result then follows from the big-step rule that corresponds with the redex.

Case 13.7 ($\mathsf{fst}\ (t, s)$, $\mathsf{snd}\ (t, s)$, $\mathsf{out}\ (\mathsf{in}\ t)$, $\mathsf{prev}\ (\mathsf{next}\ t)$, $\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ t\ \mathsf{else}\ s$, $\mathsf{if}\ \mathtt{ff}\ \mathsf{then}\ t\ \mathsf{else}\ s$)
By (Red Fst), $(\mathsf{fst}\ (t, s), w^*, [\,]) \to_n (t, w^*, [\,])$ and thus $w = 1$ and $S = [\,]$. By assumption, $t \overset{n}{\Downarrow}{}^{S'}_{w'} G$. By (Big Val), $(t, s) \overset{n}{\Downarrow}{}^{[]}_{1} (t, s)$. Then by (Big Fst), the desired result is obtained.

Case 13.8 ($\mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t$, $\mathsf{unbox}\ (\mathsf{box}\ [\vec{x} \leftarrow \vec{t}].\ t)$)
By (Red Prev), $(\mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t, w^*, [\,]) \to_n (\mathsf{prev}\ t[\vec{t}/\vec{x}], w^*, [\,])$. By assumption, $\mathsf{prev}\ t[\vec{t}/\vec{x}] \overset{n}{\Downarrow}{}^{S'}_{w'} G$. Then by (Big Box), the desired result is obtained.

Case 13.9 ($(\lambda x.\ t)\ V$)
By (Red App), $((\lambda x.\ t)\ V, w^*, [\,]) \to_n (t[V/x], w^*, [\,])$. By assumption, $t[V/x] \overset{n}{\Downarrow}{}^{S'}_{w'} G$. As V and $\lambda x.\ t$ are already values, by (Big App), $(\lambda x.\ t)\ V \overset{n}{\Downarrow}{}^{S'}_{w'} G$.

Case 13.10 ($\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}$,
$\mathsf{match}\ (\mathsf{inR}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}$)
By (Red MatchL), $(\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, w^*, [\,]) \to_n (s[V/x], w^*, [\,])$. By assumption, $s[V/x] \overset{n}{\Downarrow}{}^{S'}_{w'} G$. Then, by (Big MatchL), $\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \} \overset{n}{\Downarrow}{}^{S'}_{w'} G$.

Case 13.11 ($\mathsf{fix}\ f.\ t$)
By (Red Fix), $(\mathsf{fix}\ f.\ t, w^*, [\,]) \to_n (t[\mathsf{fix}\ f.\ t/f], w^*, [\,])$. By assumption, $t[\mathsf{fix}\ f.\ t/f] \overset{n}{\Downarrow}{}^{S'}_{w'} G$. Then, by (Big Fix), $\mathsf{fix}\ f.\ t \overset{n}{\Downarrow}{}^{S'}_{w'} G$.

Case 13.12 ($\mathsf{next}\ t \circledast \mathsf{next}\ s$)
By (Red DApp), $(\mathsf{next}\ t \circledast \mathsf{next}\ s, w^*, [\,]) \to_n (\mathsf{next}\ (t\ s), w^*, [\,])$. By assumption, $\mathsf{next}\ (t\ s) \overset{n}{\Downarrow}{}^{S'}_{w'} G$. Then, by (Big Dapp), $\mathsf{next}\ t \circledast \mathsf{next}\ s \overset{n}{\Downarrow}{}^{S'}_{w'} G$.

Case 13.13 ($\mathsf{normal}\ (\mu, \sigma,)$)
Here, there are two possibilities:

- If $(t, w^*, S) \to_n (t', w^* \cdot w, [\,])$ was derived with (Small Norm), then $S = [c]$ and $w = \mathrm{PDF}_{(\mu,\sigma)}(c) > 0$. Then, by (Big Norm), $\mathsf{normal}\ (\mu, \sigma) \overset{n}{\Downarrow}{}^{S}_{w} c$
- If $(t, w^*, S) \to_n (t', w^* \cdot w, [\,])$ was derived with (Small Fail), then $S = [c]$ and $w = \mathrm{PDF}_{(\mu,\sigma)}(c) = 0$. Then by (Big Norm Fail), $\mathsf{normal}\ (\mu, \sigma) \overset{n}{\Downarrow}{}^{S}_{w} \mathbf{fail}$

Induction step: Assume Lemma 13 holds for some $s = C[R]$ with $R \neq \textbf{fail}$, then for some $C' \neq [\cdot]$, we take $t = C'[C[R]]$. Now by case analysis on C':

Case 13.14 ($\mathsf{fst}\ [\cdot]$, $\mathsf{snd}\ [\cdot]$)

$(\mathsf{fst}\ C[R], w^*, S) \to_n (\mathsf{fst}\ C[R_1], w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$\mathsf{fst}\ C[R_1]\ ^n\!\Downarrow^{S'}_{w'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)
$C[R_1]\ ^n\!\Downarrow^{S_1}_{w_1} (\,t_1, t_2\,),\ t_1\ ^n\!\Downarrow^{S_2}_{w_2} G$	(By 2, Big Fst)	(4)
$C[R]\ ^n\!\Downarrow^{S + S_1}_{w \cdot w_1} (\,t_1, t_2\,)$	(By induction hypothesis)	(5)
$\mathsf{fst}\ C[R]\ ^n\!\Downarrow^{S + S_1 + S_2}_{w \cdot w_1 \cdot w_2} G$	(By Big Fst)	(6)

Where $S' = S_1 + S_2$ and $w' = w_1 \cdot w_2$, giving desired result.

Case 13.15 ($\mathsf{unbox}\ [\cdot]$)

$(\mathsf{unbox}\ C[R], w^*, S) \to_n (\mathsf{unbox}\ C[R_1], w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$\mathsf{unbox}\ C[R_1]\ ^n\!\Downarrow^{S'}_{w'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)
$C[R_1]\ ^n\!\Downarrow^{S_1}_{w_1} \mathsf{box}\,[\vec{x} \leftarrow \vec{t}].\ C[R_1]',$ $C[R_1]'[\vec{t}/\vec{x}]\ ^n\!\Downarrow^{S_2}_{w_2} G$	(By Big Box)	(4)
$C[R]\ ^n\!\Downarrow^{S + S_1}_{w \cdot w_1} \mathsf{box}\,[\vec{x} \leftarrow \vec{t}].\ C[R_1]'$	(By induction hypothesis)	(5)
$\mathsf{unbox}\ C[R]\ ^n\!\Downarrow^{S + S_1 + S_2}_{w \cdot w_1 \cdot w_2} G$	(By Big Box)	(6)

Where $S' = S_1 + S_2$ and $w' = w_1 \cdot w_2$, giving the desired result.

Case 13.16 ($\mathsf{prev}\ [\cdot]$)

$(\mathsf{prev}\ C[R], w^*, S) \to_n (\mathsf{prev}\ C[R_1], w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$\mathsf{prev}\ C[R_1]\ ^n\!\Downarrow^{S'}_{w'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)
$C[R_1]\ ^n\!\Downarrow^{S_1}_{w_1} \mathsf{next}\ C[R_1]',\ C[R_1]'\ ^n\!\Downarrow^{S_2}_{w_2} G$	(By Big PrevE)	(4)
$C[R]\ ^n\!\Downarrow^{S + S_1}_{w \cdot w_1} \mathsf{next}\ C[R_1]'$	(By induction hypothesis)	(5)
$\mathsf{prev}\ C[R]\ ^n\!\Downarrow^{S + S_1}_{w \cdot w_1} G$	(By Big PrevE)	(6)

Where $S' = S_1 + S_2$ and $w' = w_1 \cdot w_2$, giving the desired result.

Case 13.17 ($\mathsf{out}\ [\cdot]$)

$(\mathsf{out}\ C[R], w^*, S) \to_n (\mathsf{out}\ C[R_1], w^* \cdot w, [\,])$ for some R_1	(By assumption)	(1)
$\mathsf{out}\ C[R_1]\ ^n\!\Downarrow^{S'}_{w'} G$	(By assumption)	(2)
$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$	(By 1, Lemma 8)	(3)

$C[R_1]\ {}^{n}\!\Downarrow^{S_1}_{w_1}$ in G (By Big Out) (4)

$C[R]\ {}^{n}\!\Downarrow^{S + S_1}_{w \cdot w_1}$ in G (By induction hypothesis) (5)

fst $C[R]\ {}^{n}\!\Downarrow^{S + S_1}_{w \cdot w_1} G$ (By Big Out) (6)

Where $S' = S_1$ and $w' = w_1$, giving the desired result.

Case 13.18 (if $[\cdot]$ then t else s_0)

(if $C[R]$ then t else $s_0, w^*, S) \to_n$
(if $C[R_1]$ then t else $s_0, w^* \cdot w, [\,])$ for some R_1 (By assumption) (1)

(if $C[R_1]$ then t_0 else $s_0)\ {}^{n}\!\Downarrow^{S'}_{w'} G$ (By assumption) (2)

$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$ (By 1, Lemma 8) (3)

- If (if $C[R_1]$ then t_0 else $s_0)\ {}^{n}\!\Downarrow^{S'}_{w'} G$ was derived using (Big IteT), then

  $C[R_1]\ {}^{n}\!\Downarrow^{S_1}_{w_1}$ tt, $\quad t_0\ {}^{n}\!\Downarrow^{S_2}_{w_2} G$ (By assumption) (1)

  $C[R]\ {}^{n}\!\Downarrow^{S + S_1}_{w \cdot w_1}$ tt (By induction hypothesis) (2)

  $C[R]\ {}^{n}\!\Downarrow^{S + S_1 + S_2}_{w \cdot w_1 \cdot w_2} G$ (By Big IteT) (3)

  Where $S' = S_1 + S_2$ and $w' = w_1 \cdot w_2$, giving the desired result.

- If (if $C[R_1]$ then t_0 else $s_0)\ {}^{n}\!\Downarrow^{S'}_{w'} G$ was derived using (Big IteF), the reasoning is analogous.

Case 13.19 $((\lambda x.\ t_0)\ [\cdot])$

$((\lambda x.\ t_0)\ C[R], w^*, S) \to_n$
$((\lambda x.\ t_0)\ C[R_1], w^* \cdot w, [\,])$ for some R_1 (By assumption) (1)

$(\lambda x.\ t_0)\ C[R_1]\ {}^{n}\!\Downarrow^{S'}_{w'} G$ (By assumption) (2)

$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$ (By 1, Lemma 8) (3)

$C[R_1]\ {}^{n}\!\Downarrow^{S_1}_{w_1} V, \quad t_0[V/x]\ {}^{n}\!\Downarrow^{S_2}_{w_2} G$ (By Big App) (4)

$C[R]\ {}^{n}\!\Downarrow^{S + S_1}_{w \cdot w_1} V$ (By induction hypothesis) (5)

$(\lambda x.\ t_0)\ C[R]\ {}^{n}\!\Downarrow^{S + S_1 + S_2}_{w \cdot w_1 \cdot w_2} G$ (By Big App) (6)

Where $S' = S_1 + S_2$ and $w' = w_1 \cdot w_2$, giving the desired result.

Case 13.20 $([\cdot] \circledast t_0)$

$(C[R] \circledast t_0, w^*, S) \to_n (C[R_1] \circledast t_0, w^* \cdot w, [\,])$ for some R_1 (By assumption) (1)

$C[R_1] \circledast t_0\ {}^{n}\!\Downarrow^{S'}_{w'} G$ (By assumption) (2)

$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$ (By 1, Lemma 8) (3)

$C[R_1]\ {}^{n}\!\Downarrow^{S_1}_{w_1}$ next $C[R_1]'$,
$t_0\ {}^{n}\!\Downarrow^{s_2}_{w_2}$ next t_0' (By Big Dapp) (4)
next $(C[R_1]'\ t_0')\ {}^{n}\!\Downarrow^{S_3}_{w_3} G$

$C[R]\ {}^{n}\!\Downarrow^{S \# S_1}_{w \cdot w_1} \mathsf{next}\ C[R_1]'$ (By induction hypothesis) (5)

$C[R] \circledast t_0\ {}^{n}\!\Downarrow^{S \# S_1 \# S_2 \# S_3}_{w \cdot w_1 \cdot w_2 \cdot w_3} G$ (By Big Dapp) (6)

Where $S' = S_1 \# S_2 \# S_3$ and $w' = w_1 \cdot w_2 \cdot w_3$, giving desired result.

Case 13.21 ($\mathsf{match}\ [\cdot]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}$)

$(\mathsf{match}\ C[R]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}, w^*, S) \to_n$
$(\mathsf{match}\ C[R_1]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}, w^* \cdot w, [\,])$ for some R_1 (By assumption) (1)

$\mathsf{match}\ C[R_1]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}\ {}^{n}\!\Downarrow^{S'}_{w'} G$ (By assumption) (2)

$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$ (By 1, Lemma 8) (3)

- If $\mathsf{match}\ C[R_1]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}\ {}^{n}\!\Downarrow^{S'}_{w'} G$ was derived using (Big MatchL).

  $C[R_1]\ {}^{n}\!\Downarrow^{S_1}_{w_1} \mathsf{inL}\ V,\ t_0[V/x]\ {}^{n}\!\Downarrow^{S_2}_{w_2} G$ (By assumption) (1)

  $C[R]\ {}^{n}\!\Downarrow^{S \# S_1}_{w \cdot w_1} \mathsf{inL}\ V$ (By induction hypothesis) (2)

  $\mathsf{match}\ C[R]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}\ {}^{n}\!\Downarrow^{S \# S_1 \# S_2}_{w \cdot w_1 \cdot w_2} G$ (By Big MatchL) (3)

  Where $S' = S_1 \# S_2$ and $w' = w_1 \cdot w_2$, giving desired result.

- If $\mathsf{match}\ C[R_1]\ \{\ \mathsf{inL}\ x \mapsto t_0\ ;\ \mathsf{inR}\ y \mapsto s\ \}\ {}^{n}\!\Downarrow^{S'}_{w'} G$ was derived using (Big MatchR), the reasoning is analogous.

Case 13.22 ($\gamma(t_1, \ldots, t_{i-1}, [\cdot], t_{i+1} \ldots, t_{|\gamma|})$)

$(\gamma(t_1, \ldots, t_{i-1}, C[R], t_{i+1} \ldots, t_{|\gamma|}), w^*, S) \to_n$
$(\gamma(t_1, \ldots, t_{i-1}, C[R_1], t_{i+1} \ldots, t_{|\gamma|}), w^* \cdot w, [\,])$ for some R_1 (By assumption) (1)

$(\gamma(t_1, \ldots, t_{i-1}, C[R_1], t_{i+1} \ldots, t_{|\gamma|})\ {}^{n}\!\Downarrow^{S'}_{w'} G$ (By assumption) (2)

$(C[R], w^*, S) \to_n (C[R_1], w^* \cdot w, [\,])$ (By 1, Lemma 8) (3)

$t_1\ {}^{n}\!\Downarrow^{S_1}_{w_1} \beta_1\ \ldots\ t_{i-1}\ {}^{n}\!\Downarrow^{S_{i-1}}_{w_{i-1}} \beta_{i-1},$
$C[R_1]\ {}^{n}\!\Downarrow^{S_i}_{w_i} \beta_i,$ (By Big Sig) (4)
$t_{i+1}\ {}^{n}\!\Downarrow^{S_{i+1}}_{w_{i+1}} \beta_{i+1}\ \ldots\ t_{|\gamma|}\ {}^{n}\!\Downarrow^{S_{|\gamma|}}_{w_{|\gamma|}} \beta_{|\gamma|}$

$C[R]\ {}^{n}\!\Downarrow^{S \# S_i}_{w \cdot w_i} \beta_i$ (By induction hypothesis) (5)

$(\gamma(t_1, \ldots, t_{i-1}, C[R], t_{i+1} \ldots, t_{|\gamma|})\ {}^{n}\!\Downarrow^{S_1 \# \cdots \# S_{|\gamma|}}_{w_1 \cdots w_{|\gamma|}} G$ (By Big Sig) (6)

All cases hold and thus the induction step holds. Therefore, Lemma 13 holds by induction on the structure of t. □

B Complete equivalence proof

Proof. The left-to-right implication is proven using induction on the derivation of $t\ {}^{n}\!\Downarrow^{S}_{w} G$ (analogous cases are grouped together). Note that the induction hypothesis here assumes equivalence of the tops of the big-step rules.

Case 1.5 (Big CFail)
By induction hypothesis: $(t, 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$. Then by Red CFail, and Small Red: $(C[t], 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$

Case 1.6 (Big Norm Fail)
By assumption, $\mathrm{PDF}_{(\mu,\sigma)}(c) = 0$. Then by Small Fail: $(\mathsf{normal}\ (\mu, \sigma), 1, [c]) \Rightarrow_n (\mathbf{fail}, 0, [\,])$

Case 1.7 (Big Next Fail)
By induction hypothesis, $(t, 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$. Then by (Red NFail), $(\mathsf{next}\ t, 1, S) \Rightarrow_n (\mathbf{fail}, w, [\,])$.

Case 1.8 (Big Val)
By reflexivity (t is a value).

Case 1.9 (Big IteT,Big IteF)

$(t, 1, S_1) \Rightarrow_n (\mathtt{tt}, w_1, [\,])$, $(s, 1, S_2) \Rightarrow_n (G, w_2, [\,])$ (By Induction hypothesis) (1)

Using $C = \mathsf{if}\ [\cdot]\ \mathsf{then}\ s\ \mathsf{else}\ r$:
$(\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r, 1, S_1) \Rightarrow_n (\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ s\ \mathsf{else}\ r, w_1, [\,])$ (By Lemma 10) (2)

$(\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r, 1, S_1 +\!\!+ S_2) \Rightarrow_n (\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ s\ \mathsf{else}\ r, w_1, S_2)$ (By Lemma 11) (3)

Using $C = [\cdot]$ and (Red IfTrue)
$(\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ s\ \mathsf{else}\ r, w_1, S_2) \rightarrow_n (s, w_1, S_2)$ (By (Small Red) (4)

$(s, 1 \cdot w_1, S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\,])$ (By 1, Lemma 12) (5)

$(\mathsf{if}\ t\ \mathsf{then}\ s\ \mathsf{else}\ r, 1, S_1 +\!\!+ S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\,])$ (By 3, 4, 5) (6)

Case 1.10 (Big Next)
By induction hypothesis, $(t, 1, S) \Rightarrow_n (V, w, [\,])$. Then by Lemma 9, $(\mathsf{next}\ t, 1, S) \Rightarrow_{n+1} (\mathsf{next}\ V, w, [\,])$

Case 1.11 (Big Norm)
By the premises of (Big Norm), using (Small Norm), taking $C = [\cdot]$, $(\mathsf{normal}\ (\mu, \sigma), 1, [c]) \rightarrow_n (c, w, [\,])$.

Case 1.12 (Big PrevE)

$(t, 1, S_1) \Rightarrow_n (\mathsf{next}\ t', w_1, [\,])$, $(t', 1, S_2) \Rightarrow_n (G, w_2, [\,])$ (By induction hypothesis) (1)

Using $C = \mathsf{prev}\ [\cdot]$:
$(\mathsf{prev}\ t, 1, S_1) \Rightarrow_n (\mathsf{prev}\ (\mathsf{next}\ t'), w_1, [\,])$ (By Lemma 10) (2)

$(\mathsf{prev}\ t, 1, S_1 +\!\!+ S_2) \Rightarrow_n (\mathsf{prev}\ (\mathsf{next}\ t'), w_1, S_2)$, (By Lemma 11) (3)

Using $C = [\cdot]$ and Red PrevE $(\mathsf{prev}\ (\mathsf{next}\ t'), w_1, S_2) \to_n (t', w_1, S_2)$	(By Small Red)	(4)
$(t', 1 \cdot w_1, S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\,])$	(By 1, Lemma 12)	(5)
$(\mathsf{prev}\ t, 1, S_1 +\!\!+ S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\cdot])$	(By 3, 4, 5)	(6)

Case 1.13 (Big Fst, Big Snd)

$(t, 1, S_1) \Rightarrow_n ((\,t_1, t_2\,), w_1, [\,])$, $(t_1, 1, S_2) \Rightarrow_n (G, w_2, [\,])$	(By induction hypothesis)	(1)
Using $C = \mathsf{fst}\ [\cdot]$: $(\mathsf{fst}\ t, 1, S_1) \Rightarrow_n (\mathsf{fst}\ (\,t_1, t_2\,), w_1, [\,])$	(By Lemma 10)	(2)
$(\mathsf{fst}\ t, 1, S_1 +\!\!+ S_2) \Rightarrow_n (\mathsf{fst}\ (\,t_1, t_2\,), w_1, S_2)$,	(By Lemma 11)	(3)
Using $C = [\cdot]$ and Red Fst : $(\mathsf{fst}\ (\,t_1, t_2\,), w_1, S_2) \to_n (t_1, w_1, S_2)$	(By Small Red)	(4)
$(t_1, 1 \cdot w_1, S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\,])$	(By 1, Lemma 12)	(5)
$(\mathsf{fst}\ t, 1, S_1 +\!\!+ S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\cdot])$	(By 3, 4, 5)	(6)

Case 1.14 (Big Prev)
By induction hypothesis, $(\mathsf{prev}\ t[\vec{t}/\vec{x}], 1, S) \Rightarrow_n (G, w, [\,])$. By (Red Prev) and (Small Red), we obtain $(\mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t, 1, [\cdot]) \to_n (\mathsf{prev}\ t[\vec{t}/\vec{x}], 1, [\cdot])$. Then by Lemmas 11,12 we get $(\mathsf{prev}\ [\vec{x} \leftarrow \vec{t}].\ t, 1, S) \Rightarrow_n (G, w, [\,])$.

Case 1.15 (Big Fix)
By induction hypothesis, $(f[\mathsf{fix}\, f.\ t/f], 1, S) \Rightarrow_n (G, w, [\,])$. By (Red Fix) and (Small Red), we obtain $(\mathsf{fix}\, f.\ t, 1, [\,]) \to_n (f[\mathsf{fix}\, f.\ t/f], 1, [\,])$. Then by Lemmas 11,12 we get $(\mathsf{fix}\, f.\ t, 1, S) \Rightarrow_n (G, w, [\,])$.

Case 1.16 (Big Out)

$(t, 1, S) \Rightarrow_n (\mathsf{in}\ G, w, [\,])$	(By induction hypothesis)	(1)
Using $C = \mathsf{out}\ [\cdot]$: $(\mathsf{out}\ t, 1, S) \Rightarrow_n (\mathsf{out}\ (\mathsf{in}\ G), w, [\,])$	(By Lemma 10)	(2)
Using $C = [\cdot]$ and Red Out $(\mathsf{out}\ (\mathsf{in}\ G), w, [\,]) \to_n (G, w, [\,])$	(By Small Red)	(3)
$(\mathsf{out}\ t, 1, S) \Rightarrow_n (G, w, [\,])$	(2, 3)	(4)

Case 1.17 (Big App, Big Dapp)

$(t, 1, S_1) \Rightarrow_n (\lambda x.\ t', w_1, [\,])$, $(s, 1, S_2) \Rightarrow_n (V, w_2, [\,])$, $(t'[V/x], 1, S_3) \Rightarrow_n (G, w_3, [\,])$	(By induction hypothesis)	(1)
Using $C = [\cdot]\ s$ $(t\ \ s, 1, S_1) \Rightarrow_n ((\lambda x.\ t')\ \ s, w_1, [\,])$	(By Lemma 10)	(2)

$(t\ \ s, 1, S_1 +\!\!+ S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ \ s, w_1, S_2 +\!\!+ S_3)$ (By Lemma 11) (3)

Using $C = (\lambda x.\ t')\ [\cdot]$
$((\lambda x.\ t')\ \ s, 1, S_2) \Rightarrow_n ((\lambda x.\ t')\ \ V, w_2, [\,])$ (By 1, Lemma 10) (4)

$((\lambda x.\ t')\ \ s, 1, S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ \ V, w_2, S_3)$ (By Lemma 11) (5)

$((\lambda x.\ t')\ \ s, w_1 \cdot 1, S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ \ V, w_1 \cdot w_2, S_3)$ (By Lemma 12) (6)

$(t\ \ s, 1, S_1 +\!\!+ S_2 +\!\!+ S_3) \Rightarrow_n ((\lambda x.\ t')\ \ V, w_1 \cdot w_2, S_3)$ (By 3, 6) (7)

Using $C = [\cdot]$ and Red App
$((\lambda x.\ t')\ \ V, w_1 \cdot w_2, S_3) \rightarrow_n (t'[V/x], w_1 \cdot w_2, S_3)$ (By Small Red) (8)

$(t'[V/x], w_1 \cdot w_2 \cdot 1, S_3) \Rightarrow_n (G, w_1 \cdot w_2 \cdot w_3, [\,])$ (By 1, Lemma 12) (9)

$(t\ \ s, 1, S_1 +\!\!+ S_2 +\!\!+ s_3) \Rightarrow_n (G, w_1 \cdot w_2 \cdot w_3, [\,])$ (By 7, 8, 9) (10)

Case 1.18 (Big MatchL, Big MatchR)

$(t, 1, S_1) \Rightarrow_n (\mathsf{inL}\ V, w_1, [\,])$,
$(S[V/x], 1, S_2) \Rightarrow_n (G, w_2, [\,])$ (By induction hypothesis) (1)

Using $C = \mathsf{match}\ [\cdot]\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \} :$
$(\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, 1, S_1)$
$\Rightarrow_n (\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, w_1, [\,])$ (By Lemma 10) (2)

$(\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, 1, S_1 +\!\!+ S_2)$
$\Rightarrow_n (\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, w_1, S_2)$ (By Lemma 11) (3)

Using $C = [\cdot]$ and Red MatchL :
$(\mathsf{match}\ (\mathsf{inL}\ V)\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, w_1, S_2)$
$\rightarrow_n (s[V/x], w_1, S_2)$ (By Small Red) (4)

$(s[V/x], 1 \cdot w_1, S_2) \Rightarrow_n (G, w_1 \cdot w_2, [\,])$ (By 1, Lemma 12) (5)

$(\mathsf{match}\ t\ \{\ \mathsf{inL}\ x \mapsto s\ ;\ \mathsf{inR}\ y \mapsto r\ \}, 1, S_1 +\!\!+ S_2)$
$\Rightarrow_n (G, w_1 \cdot w_2, [\cdot])$ (By 3, 4, 5) (6)

Case 1.19 (Big Sig)

$(t_i, 1, S_i) \Rightarrow_n (\beta_i, w_i, [\,]), \quad \text{for } 1 \le i \le |\gamma|$ (Induction hypothesis) (1)

With $w_0 = 1$, and $S_{|\gamma|+1} = [\,]$, for $1 \le i \le |\gamma|$,
using $C = \gamma(\beta_1, \ldots, \beta_{i-1}, [\cdot], t_{i+1}, \ldots, t_{|\gamma|}) :$
$(\gamma(\beta_1, \ldots, \beta_{i-1}, t_i, t_{i+1}, \ldots, t_{|\gamma|}), w_0 \cdots w_{i-1}, S_i +\!\!+ \cdots +\!\!+ S_{|\gamma|+1})$
$\Rightarrow_n (\gamma(\beta_1, \ldots, \beta_i, t_{i+1}, \ldots, t_{|\gamma|}), w_0 \cdots w_i, S_{i+1} +\!\!+ \cdots +\!\!+ S_{|\gamma|+1})$ (By Lemmas 10,11,12) (2)

$(\gamma(t_1, \ldots, t_{|\gamma|}), 1, S_1 +\!\!+ \cdots +\!\!+ S_\gamma)$
$\Rightarrow_n (\gamma(\beta_1, \ldots, \beta_{|\gamma|}), w_1 \cdots w_{|\gamma|}, [\,])$ (By 2) (3)

Using $C = [\cdot]$ and (Red Sig) :
$(\gamma(\beta_1, \ldots, \beta_{|\gamma|}), w_1 \cdots w_{|\gamma|}, [\,]) \rightarrow_n (\sigma_\gamma(\overline{\beta_1}, \ldots, \overline{\beta_{|\gamma|}}), w_1 \cdots w_{|\gamma|}, [\,])$ (By Small Red) (4)

$(\gamma(t_1, \ldots, t_{|\gamma|}), 1, S_1 +\!\!+ \cdots +\!\!+ S_{|\gamma|})$
$\Rightarrow_n (\sigma_\gamma(\overline{\beta_1}, \ldots, \overline{\beta_{|\gamma|}}), w_1 \cdots w_{|\gamma|}, [\,])$ (By 3,4) (5)

All cases hold, and thus if $t \,{}^n\!\Downarrow_w^S G$, then $(t, 1, S) \rightarrow_n (G, w, [\,])$.

The right-to-left implication is proved using induction on the small-step derivation, making use of Lemma 13. We start by proving a more general version of the right-to-left implication:

$$\text{For all } w^* > 0, \text{ if } (t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,]) \text{ then } t \,{}^n\!\Downarrow_w^S G$$

The desired result is then obtained by simply taking $w^* = 1$. By induction on the derivation of $(t, w^*, S) \Rightarrow_n (G, w \cdot w^*, [\,])$:

Base case: If $(t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,])$ by reflexivity, then $t = G$, $w = 1$ and $S = [\,]$. By (Big Val), we get $t \,{}^n\!\Downarrow_1^{[]} G$, which is the desired result.

Induction step: Assume that $(t, w^*, S) \rightarrow_n (t', w', S') \Rightarrow_n (G, w \cdot w^*, [\,])$ for some (t', w', S'). Then there are w_1 and w_2, such that $(t, w^*, S) \rightarrow_n (t', w^* \cdot w_1, S') \Rightarrow_n (G, w^* \cdot w_1 \cdot w_2, [\,])$, where $w' = w^* \cdot w_1$ for some w_1 and $w = w_1 \cdot w_2$ for some w_2. The induction hypothesis then gives $t' \,{}^n\!\Downarrow_{w_2}^{S'} G$.

$t' \,{}^n\!\Downarrow_{w_2}^{S'} G$ (By induction hypothesis) (1)

$(t, w^*, S'') \rightarrow_n (t', w^* \cdot w_1, [\,])$, with $S = S'' +\!\!+ S'$ (By Lemma 1) (2)

$t \,{}^n\!\Downarrow_{w_1 \cdot w_2}^{S'' +\!\!+ S'} G$ (By Lemma 13) (3)

Thus, by induction on the multi-step derivation, noting that $S'' +\!\!+ S' = S$ and $w_1 \cdot w_2 = w$: If $(t, w^*, S) \Rightarrow_n (G, w^* \cdot w, [\,])$ then $t \,{}^n\!\Downarrow_w^S G$.
Finally, as the implication holds both ways, Theorem 1 holds. □