# Pricing composable contracts on the GP-GPU

Joakim Ahnfelt-Rønne
Michael Flænø Werk

Department of Computer Science
University of Copenhagen

August 17, 2011

**Abstract**

We present a language for specifying stochastic processes, called *SPL*. We show that *SPL* can express the price of a range of financial contracts, including so called exotic options with path dependence and with multiple sources of uncertainty. Jones, Eber and Seward previously presented a language for writing down financial contracts in a compositional manner [JES00], and specified a pricer for these contracts in terms of an abstract financial model and abstract stochastic processes. For the subset of prices that do not require nested forecasting, these can be specified in *SPL*, and we show an example of how to do this. The ease of writing a model that matches reality and the speed of computing the expected price is then highly dependent on the properties of *SPL*.

*SPL* is declarative in the sense that it is agnostic of the computational model. It is designed with the goal of matching the notation used in mathematical finance, which allows a high level specification of stochastic processes. The language is embedded in Haskell, and we have given the language formal semantics in terms of the probability monad [Gir82], as well as a type system in terms of Haskell's type system. We provide an implementation of *SPL* that performs Monte Carlo simulation on the GP-GPU, and we present data indicating that this implementation scales linearly with the number of available cores.

**Resumé**

Vi præsenterer et sprog kaldet *SPL*, hvormed man kan specificere stokastiske processer. Vi viser at *SPL* kan udtrykke prisen af en række financielle kontrakter, inklusiv såkaldte eksotiske optioner der er afhængige af værdierne i et tidsinterval og hvor adskillige kilder til usikkerhed indgår. Jones, Eber og Seward har tidligere presenteret et sprog hvori man kan skrive financielle kontrakter ved at sammensætte mindre kontrakter [JES00], og beskrevet hvordan prisen af disse fastsættes ved hjælp af en abstrakt financiel model og abstrakte stokastiske processer. Den delmængde af priser som ikke kræver indlejret estimering af fremtidige værdier kan beskrives i *SPL*, og vi viser et eksempel på hvordan dette kan gøres. Hvor nemt det er at skrive modellen så den passer til markedet og hvor hurtigt man kan udregne den forventede pris af kontrakten kommer på den måde til at afhænge meget af hvad *SPL* tilbyder.

*SPL* er deklarativt i den forstand at beregningsmodellen ikke er synlig i specifikationen af stokastiske processer. Målet har været at lægge notationen tæt op af den der bliver brugt i finansverdenen, for på den måde at gøre sproget tilgængeligt for eksperter i feltet. Sproget er defineret som et bibliotek til Haskell, og vi har givet det en formel semantik i form af sansynlighedsmonaden [Gir82], samt et typesystem i form af Haskell's typesystem. Vi viser en implementation af *SPL* der laver Monte Carlo simulering på GP-GPU'en og vi præsenterer data der indikerer at simuleringen skalerer lineært med antallet af multiprocessorer.

# Contents

# Chapter 1

# Introduction

Back in 2000, S. Jones, J-M. Eber and J. Seward [JES00] showed the benefits of writing financial contract using their domain specific language, which we will refer to as composable contracts, or *CC*. *CC* allows domain experts to compose a wide range of contract using either cash, predefined *observables* or other *CC* contract as ingredients, written in a syntax close to plain English.

The benefit of doing this is that the contracts become unambiguous by construction, which in turn makes it feasible to build applications that work for *all* of the contracts that can be expressed in the language, rather than building ad-hoc code for each new type of contract.

We will focus on one such application, namely the pricing of financial contracts. Part of this work is already done – *CC* comes with formal valuation semantics that specifies the price of any *CC* contract in the language. However, the semantics are abstract in the sense that it's given in terms of an abstract stochastic process data type and an abstract *financial model*. The stochastic processes are used to model the value of the observables and the price of the contracts, capturing the uncertainties within. The abstract model encapsulates those of the financial concepts that may need tuning to reflect the real world, such as discounting and currency exchange.

In order to provide the stochastic processes and make it possible to implement financial models, we have developed a separate domain specific language for stochastic processes called *SPL* (short for **S**tochastic **P**rocess **L**anguage) and used it to provide an implementation of the *CC* valuation semantics. The need for a separate language is due to that, although *CC* provides a way to write financial contracts, domain specific knowledge is still needed to implement the financial model and observables. Our approach to solve this problem is to allow the model and observables to be specified in *SPL*. *SPL* uses stochastic processes and distributions as basic types and is designed to look familiar to people trained in mathematical finance. We provide an example implementation of the abstract model and show how this may be used to obtain known good prices for standard financial contracts.

We have developed an OpenCL Monte Carlo simulating back end for *SPL*. Each simulation runs in complete isolation, and thus scales in a straightforward manner on the GP-GPU.

## 1.1 Background

### 1.1.1 Financial contracts and pricing

A financial contract is a set of conditions for the exchange of tradable assets between two parties - the *holder* and the *counter-party*. Commonly used tradable assets are cash and stocks but financial contracts themselves may also be traded. The condition that financial contracts are defined in terms of other tradable assets or measurable numbers has led to the names *derivative* and *underlying* where the financial contract or is called *derivative* and the tradable assets or numbers it depends on are called either *underlyings* or *observables*[1].

Financial derivatives are traded on a great scale on the world's financial markets. [Hul09] estimates the market size for derivatives to be above 600 trillions in June 2007, which is why accurate valuation methods are a major concern. The price or value of a contract is the value one would *expect* to obtain as the holder of the contract. Let's take a look at some simple contract and try to build up an intuition of how to find the corresponding expected price.

Perhaps the simplest contract is the one that immediately give the holder a certain asset. This could be the contract that pays out \$100 right away. The value of acquiring this contract is obviously \$100. But what is the value of the contract that pays out \$100 two years after the contract is engaged? This depends on the interest rate related to the dollars. Assuming a continuously compounded fixed annual risk-free interest rate of 5%, the future value of \$100 has the present value of $\$100 \cdot e^{-2 \cdot 0.05} = \$90.5$. We will have to do this kind of *discounting* whenever we are dealing with a contract that exchange assets in the future.

It is often the case that the future value of underlying assets are uncertain in the present time. As an example, this is the case when using foreign exchange or stocks as underlyings. To handle these uncertainties it is common to think of the underlyings as *stochastic processes*. A stochastic process can be seen as a function from time to a *distribution*, where the distribution describes the probability of all the possible values the underlying may have at that time. Below are two particular stochastic processes, $\mathcal{U}$ and $\mathcal{W}$:

[1] We have only seen the term *observables* used in the context of the formal contract languages [JES00, JE03].

$$
\begin{aligned}
U_{r,v,S}(t) &= Se^{(r-1/2v^2)t+v\mathcal{W}(t)} \\
\\
\mathcal{W}(0) &= 0 \\
\mathcal{W}(t+\Delta t) &= \mathcal{W}(t) + \mathcal{N}\sqrt{\Delta t}
\end{aligned}
$$

The underlying processes $U_{r,v,S}$ is defined using the other process $\mathcal{W}$ called a *Wiener process* or a *Brownian motion*. The component that makes the above two functions stochastic is the standard normal distribution $\mathcal{N}$ with mean 0 and variance 1. The time is represented as a non-negative real number representing years after the present time. Note that the plus in the Brownian motion operates on distributions as do some of the operators in the model of the underlying. We also have that the zero in the base case of the Brownian motion is representing the distribution that is certain to be zero. This kind of overloading seems to be common in finance.

The process $U_{r,v,S}$ models a standard underlying assuming again a fixed risk-free interest rate $r$ and a volatility $v$ and initial value $S$ on the underlying. The standard underlying is constructed such that the *expected* discounted value of any future value of the underlying is S, written

$$E(U_{r,v,S}(t) \cdot e^{-tr}) = S$$

Note the word *expected* in the sentence above. Whenever a contract depends on uncertain underlyings or observables, its price will become probabilistic, as modelled with the distributions. This is no good in the situation where one need to decide whether to buy a contract for a given price. This is better determined based on the *expected value* of the price distribution. The expected value of a distribution is the average of the possible values weighed by their probabilities.

Figure 1.1 shows the wild nature of a Brownian motion, which is continuous, but not differentiable at any point. It also illustrates how the underlying process $U_{r,v,S}$ depends on the Brownian motion and is influenced by the volatility.

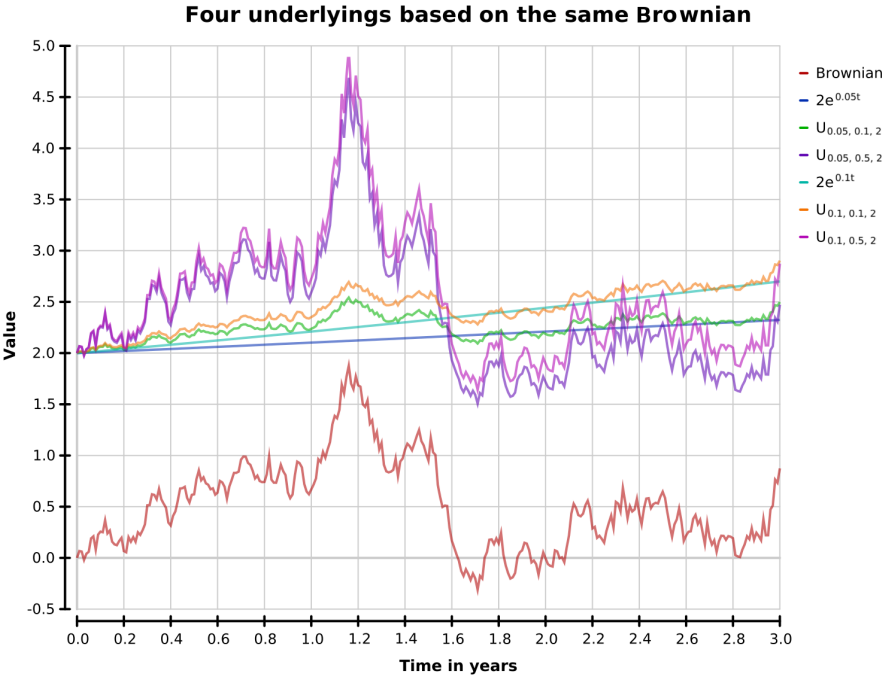



Figure 1.1: The graph shows one possible trace of the Brownian motion together with four standard underlying configurations based on the Brownian trace. The plot also shows the expected future values for the underlyings based on an interest rate of 5 and 10 percent. A large volatility creates large divergences from the expected value when the Brownian is far from zero. Larger interest rates gives a larger expected future values regardless of the Brownian value, even though this will be neglected, if the underlying is later discounted using the same rate.

Consider a more sophisticated contract - the *European call option*. This kind of contracts give the holder the *option* to buy the underlying for a fixed price, called the *strike price*, at the *maturity time* of the contract. The value of *option* contracts are never negative, as the holder is assumed to behave rationally and only exercise the option if the underlying value is larger than the strike price. This is reflected in the valuation by taking the maximum of the profit and zero. In the same way as arithmetic operators are overloaded for distributions, we assume that "pointwise" arithmetic operations are available for stochastic processes. We can write the process that describes the future value of the European call option as a function of the maturity time.

$$\text{EuropeanProcess} = \max(0,\ U_{r,v,S} - Strike)$$

In order to price the option, we only need to know the current price at a fixed maturity $t_0$, which is why we do a lookup in the process and discount the value distribution.

$$\text{EuropeanProcess}(t_0) \cdot e^{-t_0 r}$$

This method of pricing a European call option, by doing calculations directly on the stochastic processes, is illustrated in figure 1.2.

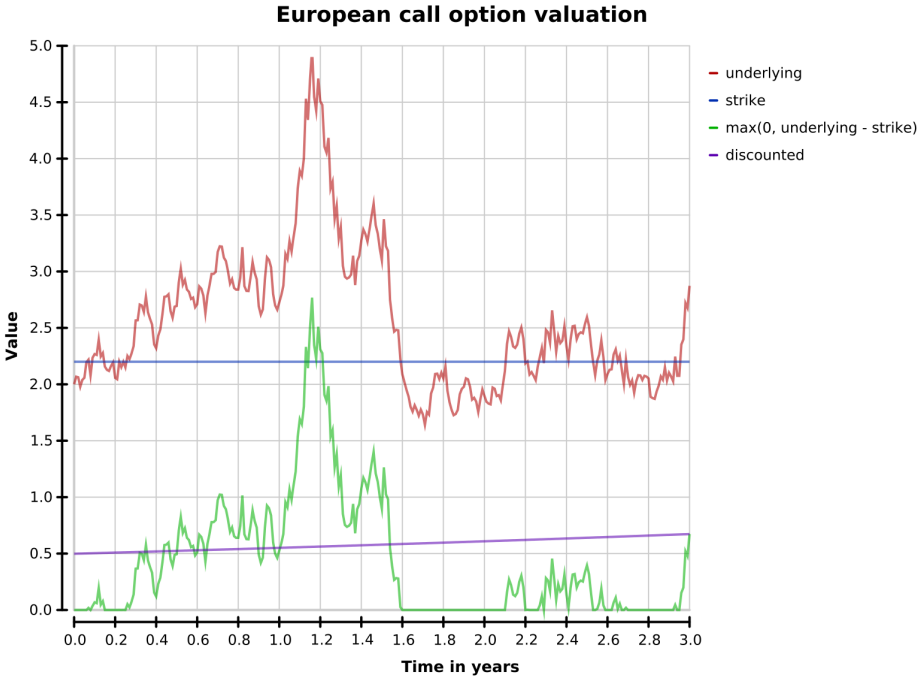



Figure 1.2: The plots in this graph illustrates the steps involved in finding the present value of an European call. The underlying process is $U_{0.1,0.5,2}$ using the same Brownian trace as in figure 1.1. The strike price is 2.2 and the maturity time is 3. The value of the European call option is discounted(0). The strike price is lifted to a process that is always certain to be the strike price. Subtracting the strike process from the underlying is simple a matter of subtracting the graphs, and the same goes for taking the maximum of the value and the zero process. The discounted process is given by: $\text{discounted}(t) = (\max(0, \text{underlying} - \text{strike}))(t_0) \cdot e^{(t-t_0)r}$, where $t_0 = 3$.

### 1.1.2 Pricing methods

In the particular future assumed in figure 1.2, the European call option would certainly have the value of 0.5. However, not being psychic, we cannot assume a particular future, but must account for all possible futures. We thus need to find the expected value of the option, which is the average value in all possible futures, weighted by their probability.

Monte Carlo simulation provides a simple way of approximating this. As done in figure 1.2 we simulate a random future by sampling from all the primitive distributions involved in the computation. This will reduce the distribution to a simple arithmetic expression on real values. We then repeat this experiment and take the arithmetic average of the simulated results and calculate the standard deviation.

While the Monte Carlo simulation method is simple, it is also rather slow. An alternative pricing technique is to describe and solve the pricing problem using partial differential equations. An example of this is the Black-Scholes method which can price European-style options using a closed formula. When such faster methods exist they are naturally preferable, but we do not have such formulas for all contracts. One of the strengths of the Monte Carlo method is that it allows pricing of contracts that are based on multiple stochastic underlyings and of contracts whose payoff depends on the path or history of the underlyings rather than just the value at payoff time [Hul09]. These conditions are present in basket options and Asian options respectively.

Another benefit of the Monte Carlo method is that each simulation is independent of the other simulations, which make the method embarrassingly parallel. This concludes our three reasons for choosing the Monte Carlo method right from the start:

- It versatile enough that it can price a wide range of financial contracts.
- It is sufficiently simple that we dare to price all these contracts, even with our limited financial knowledge.
- We can expect that an implementation will scale well on a massively parallel architecture, which would help remedy the performance issues.

### 1.1.3 Composing contracts

The two papers [JES00] and [JE03] present a domain specific language embedded in Haskell for specifying financial contract, in form of a combinator library. The following example shows a contract composed from these combinators for a particular European call option:

```
call :: Contract
call = at t0 ((underlying `and` give strikePrice) `or` zero)
```

```
    where
        strikePrice = scale 2.2 (one USD)
        underlying  = scale underlying0 (one USD)
        underlying0 = -- Observable defined elsewhere
```

The signature reveals that we have defined a contract. This is done using the `at` combinator at top level using two arguments of type `Date` and `Contract` respectively.

The date is the maturity time and the contract arguments describes the conditions for the rest of the call option, at the time of acquisition. The semantics of `at t c` is that the holder will acquire `c` at time `t`.

The sub contract `c` is, in this case, defined using `or`, which lets the contract holder choose either `c1` or `c2`. When combined with the `zero` contract, this which does nothing, this captures the *option* of acquiring a subcontract.

The *call* contract is defined using `and`, which gives the holder both of its argument contracts. In terms of the European call option, the holder will now receive both the call contract and the contract `give strikePrice`.

`one k` is the contract that pays the holder one unit in currency `k`. `scale o c` multiplies all the exchanged quantities in `c` with the value of the observable `o`. `strikePrice` is therefore the contract that pays the holder 2.2 USD.

The `give` combinator reverse the rights and obligations between the holder and the counter-party of the contract. The consequence in this case is, that `give strikePrice` obligates the holder to pay the counter-party 2.2 USD - the strike price. The reward of this is the `underlying` contract which is one USD scaled by the value of the underlying observable.

It may initially seem unnecessary involved to described a standard contract using all these tiny operators. However, the ability to build increasingly complex contracts based on simpler ones makes it possible to define a rich catalogue of contracts, and to easily extend this catalogue as needed. The user is free to define new combinators and standard contract templates as exemplified below:

```
usd :: Obs Double -> Contract
usd o = scale c (one USD)

europeanCall :: Obs Double -> Obs Double -> Contract
europeanCall u strike =
    at t0 ((usd u `and` give (usd strike)) `or` zero)
```

The *CC* combinators have also been given composable valuation semantics, which describes how to convert a *CC* contract into a stochastic process, modeling the value of the contract.

The valuation semantics of `at t c` is that of being the value of `c` at time `t` discounted to a present value, according the financial model in use. The valuation semantics of `or c1 c2` is the maximum taken on the two stochastic processes resulting from valuating `c1` and `c2`. `and` is defines likewise as the sum on the two resulting processes and `give` is negating the value of its argument. Observables in *CC* are convertible to processes, and the valuation semantics of `scale o c` is simply the meaning of multiplying `o` with the valuation of `c`. The last combinator we have seen is `one k` which exchange the value of one $k$ to the currency used in the valuation of the outer contract. This again is done according the financial model in use.

Applying these semantics to the discounted European call option from section 1.1.2 might yield the process:

$$\max(0,\ U_{r,v,S} - 2.2) \cdot e^{-t_0 r}$$

This is when the valuation is done in USD with a model using standard continuous discounting.

So why would we want to use *CC* when it seems just as easy to write the stochastic valuation processes as to write the corresponding contract? The benefits from using *CC* are many:

***CC* is declarative** The combinator `at t c` describes *that* `c` should be acquired at time `t`. Writing the corresponding price processes one would need to decide *how* the future value of `c` should be discounted, which leads to the next point.

**Replaceable financial model** There are several ways to model discounting and exchange rates which in *CC* is encapsulated in the model. This allows the users to refine or replace this model without changing the contract or to price contract using several different models.

**Contract management** A financial contract is a legal document enforcing the parties to exchange assets and should be kept somewhere. Having the legal contract formulated in *CC* eliminates the chances of a mismatch between the legal contract and the derived price process.

When a great portion of a company's assets are financial contracts, the need for a smart asset management system emerges. Such systems are provided by LexiFi[2], SimCorp[3] and others from the financial industry. It is also convenient to automate payments etc. relating to the contract. However, as previously noted the focus of this master thesis is on the pricing aspect.

[2] http://lexifi.com

[3] http://simcorp.com

**A single pricing semantics** Using the composable semantics we only need to implement the pricer once, being able to price all *CC* contracts. It is arguably easier to verify that one pricer is correct rather than having to do this verification every time a new type of contract is written.

### 1.1.4 GP-GPU and Monte Carlo simulation

Both the CUDA [NVI07] and the OpenCL [Khr08] platform provide a so called single instruction, multiple data (SIMD) architecture. As the name indicates, this is a platform where the all compute units execute the same (*single*) instructions simultaneously but are capable of using different (*multiple*) data in the calculations. Matrix addition is a simple example of a calculation suitable for the SIMD architecture. Adding two $m \times n$ arrays may be performed in parallel using $mn$ compute devices, each doing an addition, but on different elements.

The thing that makes matrix addition well suited for SIMD is its *data parallel* nature. Each scalar in the two matrices might be pairwisely isolated during the computation. Whenever we see this kind of data parallelism in a computation there is a good chance that an efficient SIMD implementation can be made. The process of distributing the data out to the compute devices and the reverse process of recombining the data will often imply some work which is not an inherent part of the original calculation. It is in general more beneficial to carry out longer distributed calculations to hide the overhead of distributing and recombining the data.

Monte Carlo simulation is a data parallel computation where the parallel data is the seeds used to simulate different results in each simulation. Using Monte Carlo simulation to find the expected value of our pricing distributions often results in rather long simulations, especially compared to the small amount of data that needs to be distributed (the seeds). As an example, consider the simulation that estimates the maximum value of a Brownian motion in one year's time. Using a time step of $1/365$ would let each parallelized simulation calculate 365 normally distributed random values.

The only real problem we seem to have left is the stage after the individual simulations have finished, where the results should be combined to an average value and a standard deviation. We will look into that in chapter 6.

### 1.1.5 The problem

As we have already seen, *CC* expresses prices as stochastic processes. However, the stochastic processes themselves, as well as the financial model, are left abstract, except from the interface to them required by the pricer. Figure 1.3 gives an overview of these interfaces. The abstract specifications give us the freedom to choose our own implementations, which in the case

of the financial model needs to be highly tuned to match the real world. To computer scientists, the mathematics behind constructing a financial model likely seems rather arcane; conversely, the details of constructing a computationally feasible implementation of such a financial model may seem rather arcane to the quantitative analysts. Ideally then, we would like to separate these two concerns so that each can be handled by the experts in their domain.

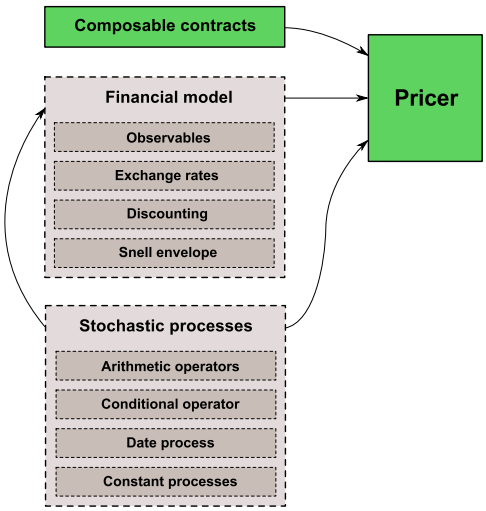



Figure 1.3: The components of *CC* relevant to pricing. The articles provides a representation of the composable contracts and an implementation of the abstract pricing algorithm (both green). The pricer also requires a financial model and a representation of stochastic processes (both with dashed lines), both of which are abstract in the definition of the pricer. The requirements of the interfaces for these are shown as inner boxes. The arrows connect components to their usage site.

Since path dependent prices are especially well suited to Monte Carlo simulation, we will use these as a show case, even though they are not currently supported by the *CC* library.

## 1.2 Our solution

We have seen in figure 1.3 that stochastic processes and a financial model is required in order to use the pricer for composable contracts. Since the financial model also requires stochastic processes, specifying these is a natural place to begin.

*CC* already defines an interface for stochastic processes. However, the implementation of the financial model will require more than is provided by this interface; for example, there is no way to introduce the uncertainty required for modelling observables.

There are also a range of options that cannot be expressed in *CC*, such as contracts whose payoff depends on the history of values of the observable. If *CC* is to be extended to cover these cases, it will require a richer set of stochastic processes.

As we are not experts in the financial domain, there is little point in trying to build a realistic financial model. Instead, it should be made feasible for domain experts to implement and experiment with such models. In order to achieve this, we need some way to write down stochastic processes that reflects the notation used in finance, and which is simultaneously computationally feasible.

This is indeed our main contribution: a domain specific language for writing down stochastic processes, and an implementation for it that does Monte Carlo simulation on massively parallel hardware. We call this language *SPL*, short for stochastic process language.

The point of domain specific languages is to make it easy to express domain specific knowledge. For example, recall the definition of the standard underlying:

$$Se^{(r-\frac{1}{2}v^2)t+v\mathcal{W}(t)}$$

We can write this stochastic process down in SPL quite effortlessly:

```
s * exp ((r - 0.5 * v^2) * time + v * brownian)
```

Like the underlying, this is a specification of a stochastic process. It's defined in terms of the simpler `brownian` and `time` processes, which are in turn defined via lower level primitives that we shall see in chapter 7.

One of the types of contract we would like to price is Asian options, whose payoff is defined as the average over a time interval. Like `brownian` and `time`, the cummulative running average is defined in the prelude for SPL, so given an `underlying` process we can simply say:

```
average underlying
```

An option would typically require the holder to pay a premium when exercising the option:

```
average underlying - exercisePrice
```

And since the holder is assumed to be rational, she will never choose to exercise if the payoff becomes negative:

```
max_ 0 (average underlying - exercisePrice)
```

Whatever the payoff is, it's not worth as much to get the money sometime in the future as it is to get it instantly. We therefore need to factor in the discounting $e^{-tr}$:

```
exp (-time * r)
```

Finally, we need to decide on an exercise date, in essence looking up the payoff at the exercise time in the stochastic process. This lands us at the final SPL definition:

```
asian underlying rate exercisePrice exerciseTime = payoff
    where
        payoff   = lookup exerciseTime process
        process  = discount * option
        discount = exp (-time * rate)
        option   = max_ 0 (average underlying - exercisePrice)
```

Looking up into a stochastic process yields a distribution – in this case the distribution of all possible payoffs. In order to make a decision on whether or not to buy the option, we need to know the payoff on average.

To be confident in the result, we want to run a lot of simulations, and we thus compile our function for the GP-GPU with an appropriate time step, say `1/365`. We can delay our decision on the parameters that are simple values, to just before the GP-GPU kernels are invoked. Let's assume that we can model the price of crude oil by giving appropriate parameters to the underlying as defined earlier:

```
> asianOilPrice <- compile (1/365) (asian oil)
```

This yields a function `asianOilPrice` that when given the remaining arguments for the partially applied function `asian`, namely the rate, exercise price and exercise time, yields the average payoff and the standard deviation:

```
> asianOilPrice 0.05 100 1
5.03 ± 7.45
```

Note that the numbers printed here are purely for illustration – we will price Asian options and other types of financial contracts with realistic parameters in chapter 9.

*SPL* is, like *CC*, a DSL embedded in Haskell. The code we have just seen is thus simply Haskell code using a library. We reuse the flexible syntax and type system of Haskell to provide the syntax and type system of our domain specific language. There is thus less work to be done to provide an implementation and less chance of introducing errors into it.

## 1.3 Results

We have designed an embedded domain specific language for continuous stochastic processes called SPL (chapter 4 and 7). We have given it formal semantics in terms of the probability monad, and used Haskell's type system to provide type safety.

We have provided an implementation of the language that performs Monte Carlo simulation on the GP-GPU (chapter 8), and whose performance scales linearly with the number of available processing elements (chapter 10).

We have implemented and tested pricers in SPL for European-style options, including European options, Asian options, Lookback options and Basket options (chapter 9). Some of these are examples of path dependent contracts and contracts with multiple sources of uncertainty which are difficult to price without Monte Carlo simulation [BBG97]. Although we have not tested Barrier options, we speculate that we could also price those, since we support path dependence and conditional logic. *CC* has no construct for path dependence, and SPL can thus price contracts that cannot be expressed in *CC*.

We can't price American/Bermudan-style options (section 7.2). When making a choice that depends on the future value of a process, the current semantics of SPL assumes that we can see into the future and determine which of all possible events will occur. Assuming the non-existence of psychics, what is needed instead is the *expected value*. We are thus unable implement the pricer for this set of *CC* financial contracts.

We have provided an implementation of the *CC* pricer in terms of SPL, as well as an example financial model (section 7.2). It is incomplete due to the issue with American/Bermudan-style options.

Our approach has been inspired by multiple domain specific languages including *CC* (chapter 3), probabilistic functional programming libraries (chapter 5) and array languages (chapter 6), and we have investigated these in the context of pricing financial contracts.

## 1.4 Acknowledgements

This master thesis would not have been possible without the generous guidance from multiple members of the HIPERFIT research center. In particular, Mogens Steffensen (IMF), Carl Balslev Clausen (SimCorp), Martin Elsman (SimCorp), Dirk Bangert (Bangert Research) and Rolf Poulsen (IMF) all helped us understand the financial concepts required to produce this thesis. We wish to thank our supervisor, Ken Friis Larsen, who suggested the subject and provided a large amount of input and feedback. Jørgen Thorlund Haahr, Ramon Soto Mathiesen and Manijeh Elsa Modi all provided valuable feedback on drafts. Finally, David B. Thomas, Geoffrey Mainland,

Manuel M. T. Chakravarty and Martin Dybdal were all very helpful in providing early access to and answering inquiries about their libraries.

## 1.5 Preliminaries and notation

We assume that the reader is familiar with lambda calculus and the programming languages Haskell [Jon02] and C [ISO99]. Any major extensions used will be explained along the way. We will occasionally show an interaction with an interactive Haskell interpreter, with the input marked by a `>` and the output following immediately after.

# Chapter 2

# Common financial contracts

This chapter describes the types of financial contracts that will be used in this thesis and points out to what degree they are expressible in *CC* and priceable in *SPL*. There are many types of financial contracts, but we are only going to focus on some of the common ones. The contracts are all derivatives expressed in terms of one or more underlying assets. Note that the naming convention is rather arbitrary, and has no actual geographical implications.

**European call options** gives you the option at a single pre-determined point in time, called the *exercise time* or *maturity time*, to buy the underlying asset for a fixed price, called the *strike price*.

**American call options** gives you the option at any point during a time interval to buy the underlying for a fixed price.

**Bermudan call options** gives you the option at several points in time to buy the underlying asset for a fixed price. The name alludes to being somewhere between European and American options.

**Asian call options** comes in at least two varieties, fixed strike and floating strike. *Fixed strike* gives you the option at a single pre-determined point in time[1] to receive the difference between the strike price and the underlying's *average* price over a pre-determined period of time. *Floating strike* gives you the option at a single pre-determined point in time to buy the underlying for the average price over a pre-determined period of time, possibly multiplied by some constant.

**Lookback call options** are like Asian call options, except that the *maximum* or *minimum* is used rather than an average.

**Put options** are similar to call options, but instead of giving you the option to buy the underlying, they give you the option to *sell* it.

[1] At least in European-style options.

**Basket options** are options that depend on multiple, possibly correlated, underlyings, such as by their weighted sum or average. These are also known as *rainbow options*.

**Barrier options** are options that either begin or expire based on events of the underlying observable, such as its price surpassing or dropping below a certain threshold.

The Asian, Lookback, Barrier and Basket options are so called *exotic* options, whereas European and American options are called *vanilla* options because they depend only on the value of a single observable and only at exercise time.

Now let us take quick look into whether these contract are expressible in *CC* and priceable *SPL*.

The European options are simple to express in *CC* and are priceable in *SPL*. There exist closed formulas to do the same, at least for the most common underlyings. We take advantage of this to test the correctness of our pricer in section 9.3.

It is also simple to express American options in *CC* as the language have specific features for these. But the pricing of American options is somewhat more involved compared to the European-style and are therefore not priceable in *SPL*. We will return to this problem in the end of section 7.2.

The Bermudan option provides the contract holder with several exercise times, but not a continuous period of time as in the American option. This is harder to express in *CC* than the American option as one would need to model this using a nesting as deep as the number of exercise times.[2] We cannot price Bermudan options for the same reason that we cannot price American options as will be pointed out in section 7.2.

Asian, Lookback and Barrier options are all *path dependant*, in that they depend on the underlying value on times other than the exercise time. We can price these, and in general, these are hard to price without resolving to Monte Carlo simulation [BBG97], and are thus ideally suited to showcase our implementation of *SPL*.

The Basket options rely on multiple underlyings which is not a problem to express in either *CC* or *SPL*.

[2]The newer library[JE03] can express this without nesting if the exercise times can be captured in a expression such as $t \ mod \ 30 = 0$.

# Chapter 3

# Composable contracts

In the introduction, we saw an example of using the *CC* combinators to define a financial contract and argued why it is beneficial to use combinators to do so. The specific set of combinators we refer to when writing *CC*, are the ones defined in [JES00]. Peyton and Eber wrote a followup on this paper [JE03], presenting a slightly modified set of contract combinators. We explain in the end of this chapter why we chose to use the older combinators. This is after a somewhat detailed look at the original paper from 2000.

## 3.1 Concepts and terminology

A *CC* contract is composed using four ingredients:

**Observables** provides a way to bring numbers from the real world into the contracts. As the contracts must be legally enforceable, observables are limited to the kind of quantities that we can agree on how to measure. This could the temperature in Paris, the 1-month LIBOR rate[1] or simply a real number. Their value is per definition known in the present, but the future value of observables are in general unknown.

**Currencies** are used to express the currency of the assets traded. The observables cannot inject numbers directly into a contract but only scale numbers in existing contracts, which in the end must be given in a specific currency. Every contract that has a value different from zero is therefore expressed in a currency.

**Times** are used to model the horizon or expiry time of a contract, which are not only used to end the contract, but also to specify when certain actions should take place. We will look into that shortly.

[1]The LIBOR (London Interbank Offered Rate) is a reference rate published daily by the British Bankers' Association

**Sub-contracts** is the last recursive ingredient, which is an important part of the composability.

The conditions specified in a *CC* contract are first realized after the contract is **acquired**. This makes the **acquisition time** important, as the conditions taking place before that time are discarded. The conditions referred to here, consist of the *obligations*[2] to receive or pay cash to the other party of the contract as well as to make decisions imposed by the contract.

The **horizon** of a contract is the time the contract will expire. The horizons are not a present part of the *CC* AST but a static feature determined by the function $H$ given in figure 7 in [JES00] or the overview table in section 3.3. A contract cannot be acquired after the horizon but an acquired contract may give the holder a new contract with a new horizon that extends the horizon of the original contract. A contract may never expire in which case we say that the horizon is infinite, reflected in the type below:

$$H : \texttt{Contract} \to \mathcal{D}_{\mathcal{AT}\mathcal{E}} \cup \{\infty\}$$

## 3.2 Implementing the abstract pricer

The compositional valuation semantic given in figure 4 in [JES00] may also be seen as an abstract implementation of the valuator or pricer. It is abstract in the sense that it presumes the presence of a financial model[3] and a data type for stochastic processes. The model and this data type needs to be implemented, in order to get a fully functional pricer implementation as illustrated in figure 1.3 from the introduction.

The stochastic process data type should conceptually model a function from time to a distribution:

$$\mathcal{P}_{\mathcal{R}}\ \texttt{a} : \mathcal{D}_{\mathcal{AT}\mathcal{E}} \to \mathcal{D}_{\mathcal{IST}}\ \texttt{a}$$

This does not necessarily mean that it should be implemented as a function but it should provide the apply or lookup operation, which we here denote using standard mathematical function notation. The process data type should also come along with standard arithmetic operations, conditionals[4] and two constructors:

[2]The original article[JES00] referred to the condition of having to receive cash or make a choice as a *right* an having to pay as a *obligation*. We refer to both of these conditions as *obligations* as the party involved is not left with a choice.

[3]The paper in discussion actually use a slightly different definition of the concept "financial model" as it includes the stochastic process primitives in the definition but not the observables. We prefer to think of the observables as part of the model, but not the stochastic process data type.

[4]This is not a explicit requirement in the original paper, but the side condition in figure 4 imposes this or a similar requirement.

$$
\begin{aligned}
&\ominus \ : \ \mathcal{P}_{\mathcal{R}}\ a \to \mathcal{P}_{\mathcal{R}}\ b \\
&\oplus \ : \ \mathcal{P}_{\mathcal{R}}\ a \to \mathcal{P}_{\mathcal{R}}\ b \to \mathcal{P}_{\mathcal{R}}\ c \\
&\texttt{if} \cdot \texttt{then} \cdot \texttt{else} \ : \ \mathcal{P}_{\mathcal{R}}\ \mathbf{B} \to \mathcal{P}_{\mathcal{R}}\ a \to \mathcal{P}_{\mathcal{R}}\ a \to \mathcal{P}_{\mathcal{R}}\ a \\
&\mathcal{K} \ : \ a \to \mathcal{P}_{\mathcal{R}}\ a \\
&\mathit{time} \ : \ \mathcal{D}_{\mathcal{ATE}} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R}
\end{aligned}
$$

$\mathcal{K}(a)$ promotes the constant $a$ to the stochastic processes always certain to be $a$ and *time* is the process always yielding the current time. This API is specified in figure 6 in the article.

The model $\mathcal{M}$ should first of all implement the three functions for doing discounting, exchange rates and to calculate the so called snell envelope which is used to price American style options where the holder have the right to exercise the option in a continuous *period* of time.

$$
\begin{aligned}
&\mathit{dics}_k^t : \mathcal{D}_{\mathcal{IST}}\ \mathbf{R} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R} \\
&\mathit{exch}_k : \mathcal{C}_{\mathcal{URRENCY}} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R} \\
&\mathit{snell}_k^t : \mathcal{P}_{\mathcal{R}}\ \mathbf{R} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R}
\end{aligned}
$$

The valuation semantic uses the function shown below to convert *CC* observables into values of the abstract stochastic process data type.

$$\mathcal{V}[\![] : \texttt{Obs a} \to \mathcal{P}_{\mathcal{R}}\ \texttt{a}$$

Figure 5 in the paper provides a partial implementation of this function, which allow the contract designer to construct constant observable values and do arithmetic on the observables corresponding to what is required for the process data type. But these constructors and operations cannot introduce the stochastic elements of observables and $\mathcal{V}[\![]$ still lacks support for the observables modelling real world numbers.

This is the last thing that is needed in the model. A function that converts the remaining real world or stochastic observables supported by the model into the process data type. As these observables are bound to measure numbers, this function should have the type $\texttt{Obs Real} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R}$. This should finish the implementation of $\mathcal{V}[\![]$ and we now have a fully functional implementation of the valuation function, provided all the functionality mentioned above.

$$\mathcal{E}_k[\![] : \texttt{Contract} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R}$$

The valuation function converts a *CC* contract into the process specifying the price of the contract over time, expressed in currency $k$. It really is parametric of the model $\mathcal{M}$, but we will continue to use the notion from the two papers, where this is not present in the notation.

## 3.3 The combinators

The valuation semantics are defined in the original paper using side conditions that *limits the domain* of the resulting price process. For `and` this is done as

$$\begin{array}{rcll} \mathcal{E}_k[\![\mathtt{c_1}\text{`}\mathtt{and}\text{`}\mathtt{c_2}]\!] & = & \mathcal{E}_k[\![\mathtt{c_1}]\!] + \mathcal{E}_k[\![\mathtt{c_2}]\!] & \text{on } \{t \mid t \leq H(\mathtt{c_1}) \wedge t \leq H(c_2)\} \\ & & \mathcal{E}_k[\![\mathtt{c_1}]\!] & \text{on } \{t \mid t \leq H(\mathtt{c_1}) \wedge t > H(c_2)\} \\ & & \mathcal{E}_k[\![\mathtt{c_2}]\!] & \text{on } \{t \mid t > H(\mathtt{c_1}) \wedge t \leq H(c_2)\} \end{array}$$

Recall that the stochastic process data type conceptually is a function from time to a distribution. The definition above defines the price process for ($\mathtt{c_1}$`and`$\mathtt{c_2}$) in terms of three processes each defined with their own time domain. These processes need to be combined into a single process. This is why we suggested the `if-then-else` operator to be part of the abstract stochastic process data type.

The overview table, given shortly, explicitly uses the `if-then-else` operator, but we will use the definition below to ease the notation

$$\mathcal{A}_k(\mathtt{c}, v) = \mathtt{if}\ time \leq \mathcal{K}(H(\mathtt{c}))\ \mathtt{then}\ \mathcal{E}_k[\![\mathtt{c}]\!]\ \mathtt{else}\ \mathcal{K}(v)$$

$\mathcal{A}_k(\mathtt{c}, v)$ is like $\mathcal{E}_k[\![\mathtt{c}]\!]$ except that it becomes $\mathcal{K}(v)$ when `c` expires. This allow us to write the valuation semantic for the `and` combinator as

$$\mathcal{E}_k[\![\mathtt{c_1}\ \text{`}\mathtt{and}\text{`}\ \mathtt{c_2}]\!] = \mathcal{A}_k(\mathtt{c_1}, 0) + \mathcal{A}_k(\mathtt{c_2}, 0)$$

This definition is not completely right as $\mathcal{E}_k[\![\mathtt{c}]\!]$ should be undefined for times after $H(c)$. We will not be explicit about this in the table below, but this should be a concern for one implementing $\mathcal{E}_k[\![]\!]$.

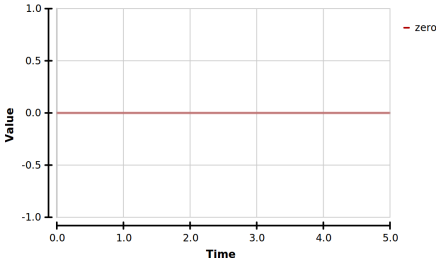
The table below gives an overview of all the *CC* combinators. Each combinator is shown together with its type, a short description, its horizon and valuation semantic, and a plot illustrating how the valuation process could look like, compared to the value of its arguments.

`zero :: Contract`

This contract imposes no obligations on any of the parties. Its price is therefore zero at all times.

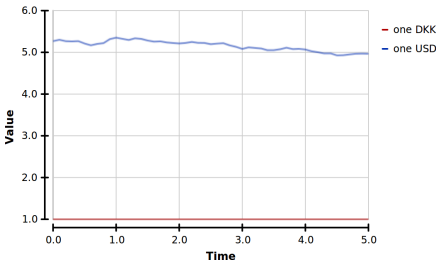
$H(\mathtt{zero}) = \infty$

$\mathcal{E}_k[\![\mathtt{zero}]\!] = 0$



### `one :: Currency -> Contract`

`one` $\mathtt{k}_2$ pays the holder one unit of `k`. The value of this contract expressed in currency `k` is the exchange rate between `k` and $\mathtt{k}_2$ captures by the model function *exch*. The *exch* used in the plot to the right uses some volatility in its model. The contracts are priced in DKK.

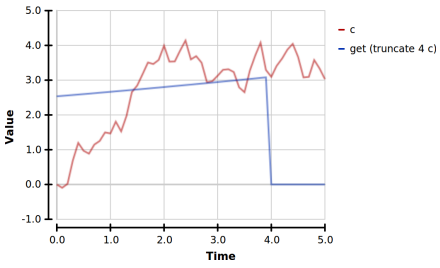



$H(\mathtt{one}\ \mathtt{k}_2) = \infty$

$\mathcal{E}_k[\![\mathtt{one}\ \mathtt{k}_2]\!] = exch_k(k_2)$

### `get :: Contract -> Contract`

`get c` let the holder acquire `c` at time $H(\mathtt{c})$. Nothing happens until then. The values of the contract is the discounted expected value at $H(\mathtt{c})$ as illustrated to the right.




$H(\mathtt{get}\ \mathtt{c}) = H(\mathtt{c})$

$\mathcal{E}_k[\![\mathtt{get}\ \mathtt{c}]\!] = dics_k^{H(c)}(\mathcal{E}_k[\![\mathtt{c}]\!](H(c)))$
**if** $H(c) \neq \infty$

### `anytime :: Contract -> Contract`

Acquiring `anytime c` at time $t$ obligates the holder to acquire `c` between $t$ and $H(\mathtt{c})$. Our current implementation of *SPL* is not expressive enough to implement the model function *snell*, which is why no plot is provided.

$H(\mathtt{anytime}\ \mathtt{c}) = H(\mathtt{c})$

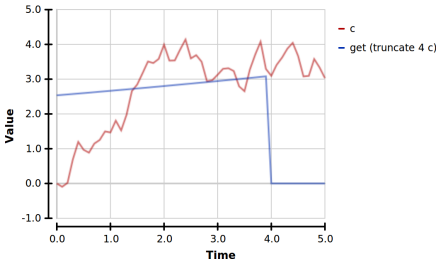
$\mathcal{E}_k[\![\mathtt{get}\ \mathtt{c}]\!] = snell_k^{H(c)}(\mathcal{E}_k[\![\mathtt{c}]\!])$
**if** $H(c) \neq \infty$

### `truncate :: Date -> Contract -> Contract`

`truncate t c` is like `c` except that the horizon may have been expedited.

$H(\texttt{truncate t c}) = min(t, H(\texttt{c}))$

$\mathcal{E}_k[\![\texttt{truncate t c}]\!] = \mathcal{E}_k[\![\texttt{c}]\!]$

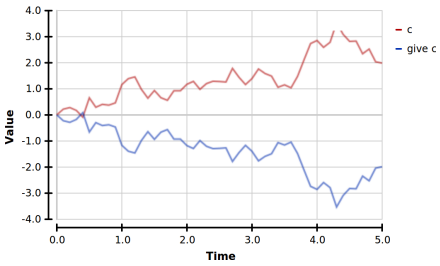



### `give :: Contract -> Contract`

`give c` is equivalent to `c` where the obligations have been swapped for the two parties. This is why the value of `give c` is the negated value of `c` as illustrated to the right.

$H(\texttt{give c}) = H(\texttt{c})$

$\mathcal{E}_k[\![\texttt{give c}]\!] = -\mathcal{E}_k[\![\texttt{c}]\!]$

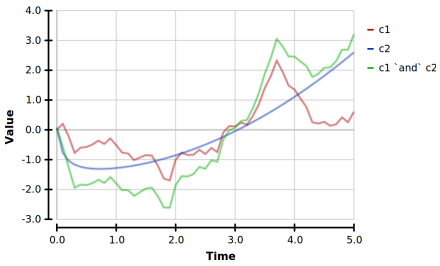



### `and :: Contract -> Contract -> Contract`

When acquiring $\texttt{c}_1$ ‘`and`‘ $\texttt{c}_2$ the holder acquire *both* $\texttt{c}_1$ *and* $\texttt{c}_2$ except the ones that are expired.

$H(\texttt{c}_1\ \text{‘}\texttt{and}\text{‘}\ \texttt{c}_2) = max(H(\texttt{c}_1), H(\texttt{c}_2))$

$\mathcal{E}_k[\![\texttt{c}_1\ \text{‘}\texttt{and}\text{‘}\ \texttt{c}_2]\!] = \mathcal{A}_k(\texttt{c}_1, 0) + \mathcal{A}_k(\texttt{c}_2, 0)$

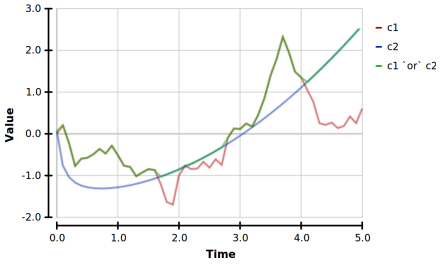



### `or :: Contract -> Contract -> Contract`

When acquiring $\texttt{c}_1$ ‘`or`‘ $\texttt{c}_2$ the holder must acquire *either* $\texttt{c}_1$ *or* $\texttt{c}_2$ except the ones that are expired.

$H(\texttt{c}_1\ \text{‘}\texttt{or}\text{‘}\ \texttt{c}_2) = max(H(\texttt{c}_1), H(\texttt{c}_2))$

$\mathcal{E}_k[\![\texttt{c}_1\ \text{‘}\texttt{or}\text{‘}\ \texttt{c}_2]\!] =$
$max(\mathcal{A}_k(\texttt{c}_1, -\infty), \mathcal{A}_k(\texttt{c}_2, -\infty))$

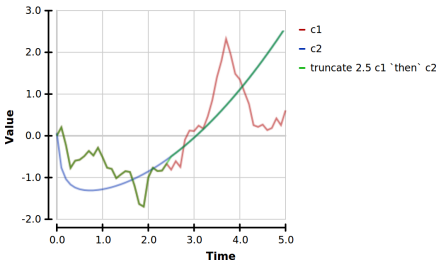


`then :: Contract -> Contract -> Contract`

$c_1$ ‘`then`‘ $c_2$ lets the holder acquire first $c_1$ and *then* $c_2$ at time $H(c_1)$, if $H(c_2) > H(c_1)$. If $c_1$ has expired at acquisition time of ($c_1$ ‘`then`‘ $c_2$) then $c_2$ must be acquired.

$H(c_1 \text{ `then` } c_2) = max(H(c_1), H(c_2))$

$\mathcal{E}_k[\![c_1 \text{ `then` } c_2]\!] =$
`if` $time < \mathcal{K}(H(c_1))$
`then` $\mathcal{E}_k[\![c_1]\!]$ `else` $\mathcal{E}_k[\![c_2]\!]$

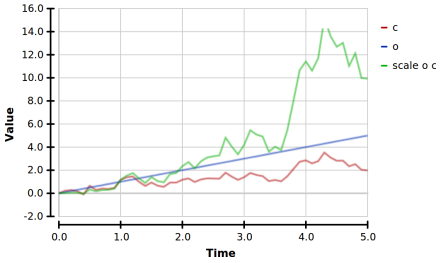



`scale :: Obs Double -> Contract -> Contract`

`scale o c` is equivalent to `c` where all the payments have been multiplied by `o`.

$H(\texttt{scale o c}) = H(\texttt{c})$

$\mathcal{E}_k[\![\texttt{scale o c}]\!] = \mathcal{V}[\![\texttt{o}]\!] * \mathcal{E}_k[\![\texttt{c}]\!]$




## 3.4 The two versions

As mentioned in the beginning of this chapter there are two different versions of the *CC* combinators. The one we are using are those described in the paper from 2000 [JES00] and the other version mentioned, is the combinators from the 2003 paper [JE03]. We will here look at how the new version mainly extends the expressiveness of the older version and then return to cost of this, seen in an implementation context. The 2003 paper is very sparse in its comparison of the new reviewed library to the older one and does not mention the extra implications involved in implementing the new combinators as discussed below.

The 2003 version removes the four combinators `truncate`, `then`, `get` and `anytime` from the 2000 version but introduces four new, shown below. The 2003 combinators are typeset in italic.

$$
\begin{array}{lcl}
\textit{cond} & :: & \textit{Obs Bool} \rightarrow \textit{Contract} \rightarrow \textit{Contract} \rightarrow \textit{Contract} \\
\textit{when} & :: & \textit{Obs Bool} \rightarrow \textit{Contract} \rightarrow \textit{Contract} \\
\textit{anytime} & :: & \textit{Obs Bool} \rightarrow \textit{Contract} \rightarrow \textit{Contract} \\
\textit{until} & :: & \textit{Obs Bool} \rightarrow \textit{Contract} \rightarrow \textit{Contract}
\end{array}
$$

These are all combinators that deal with time, but this is no longer done

using the horizon but using the Boolean observable argument. Say we want to write the contract where $c$ is acquired at time 5, this could be done by *when* $(time == 5)$ $c$, using the new combinators. This corresponds to `get (truncate 5 c)` using the old library. Using the new combinators, it is no longer valid to talk about the horizons or expiry time. The horizons served at least two purposes in the old combinator definition. First, they provided a mechanism for letting an ongoing contract come to an end. Secondly, they provided the combinators `then`, `get` and `anytime` with information, thereby serving as an alternative to an extra argument.

The first usage mentioned are captured in the 2003 version by the combinator *until* $o$ $c$, that will abandon the contract at the first time $o$ becomes true. But *until* is even stronger than that as the abandon-time is not a constant but an observable. This could for instance allow one to implement the *exotic* option type where the option is cancelled if the underlying breaches a certain barrier[5].

The second usage of horizons in the 2000 library is captured in the 2003 version by replacing the combinators that relied on the horizon with new ones. *anytime* is one of these which is now taking a Boolean observable at its first argument. Where the old `anytime c` required the holder to acquire `c` at anytime up until expiry, the new *anytime* $o$ $c$ allows the holder to acquire $c$ at times where $o$ is true. This allows the exercise times to be something more complicated than just a single time period. The new version of *anytime* does not require the holder to acquire the sub contract eventually and it seems impossible to model a latest time where $c$ *must* be acquired. This is not a problem when modelling American options, but it would be if we actually wanted the old semantics of `anytime c`, where the holder must acquire `c` eventually.

*when* $o$ $c$ on the other hand does not have this problem as $c$ must be acquired as soon as $o$ becomes true. This combinators is therefore a stronger variant of the older `get`.

The last new combinator *cond* $o$ $c_1$ $c_2$ choose $c_1$ if $o$ is true at acquisition time and $c_2$ otherwise. This combinator may be seen as a variant of `then` from the old library, but it is stronger as is allows contract to model conditionals dependent on observables.

We have seen that the 2003 library version, for the most part, are more expressive than the older version. This new expressiveness comes from the new conditional combinator and because time dependent combinators have been changed from using a static constant time to using a dynamic uncertain varying time. But this flexibility comes at a price when implementing the valuation semantic for *when*, *anytime* and *until*. The pricing implementation of these is delegated to the model functions $disc_k$, $snell_K$ and the new function $absorb_k$. These three functions share the same signature

[5]This is called a up-and-out barrier option

$$\mathcal{P}_{\mathcal{R}}\ \mathbf{B} \times \mathcal{P}_{\mathcal{R}}\ \mathbf{R} \to \mathcal{P}_{\mathcal{R}}\ \mathbf{R}$$

Compared to the old model interface we now have a stochastic process of Booleans where we had a static time before. This is problematic when doing discounting as we need to know the time in which we should discount from, either as a contract or as a distribution. To find this distribution we need to scan over the process $o$ and for each time $t$ note the probability of $o(t) \wedge (\forall t' < t : \neg o(t'))$. This scan property states that the Boolean observable is true for the first time. The problem is that this does not terminate as we need to know these probabilities for infinitely large values of $t$. We might however stop the search in the special case where there exists a $t$ for which $o(t)$ is certain to be true as this would make the scan property false for any larger $t'$, but the search is still infinite in the general case. The 2003 paper gives an example of calculating $disc_{\mathcal{L}}(time = 3, \mathcal{K}(10))$. But this example utilizes the fact that $\mathcal{V}[\![\texttt{time} = \texttt{3}]\!](3)$ is certain to be true.

Implementing $absorb_k$ yields the exact same problems and we do not have a clear picture of all the challenges that would emerge when implementing the new *snell* compared to the old version.

The implementation difficulties discussed in this section, is the reason why we chose to use the old composing contract library. But even if we did find a solution to the termination problem, there are still some factors that makes the new library less suited for our choice of numerical model, using Monte Carlo simulation running on a GP-GPU. Using the Monte Carlo setting, we could imagine each thread on the GP-GPU computing the discounted value of a brownian motion, at time $t$. Doing this computation, each thread would be running a loop going though the iteratively defined brownian process, until time $t$ is reached. This is efficient on a GP-GPU as all the threads are running the same amount of loops. But had $t$ been stochastic, we would have had the case where all the threads would need to wait until the last one had finished. By having a contract language that limits all the lookup times to be non stochastic, we know that all lookups will run efficiently in parallel.

We have the same concern regarding *when*. This combinator expresses a condition based on a stochastic value. This would split the GP-GPU simulations threads into different code branches which, on the GP-GPU architecture implies, that all threads run both branches, performance wise.

# Chapter 4

# Goals for a stochastic processes language

As we have seen in chapter 3, we need to provide a language for stochastic processes in order to implement the pricer. At the very least, pointwise arithmetic operators are required, but many other features are implicitly required in order to make implementing the financial model feasible. In this chapter, we'll attempt to identify these requirements.

## 4.1 Matching the domain

Pricing financial contracts is a subtle process and is best left to the domain experts, namely quantitative analysts. However, the concepts from mathematical finance, such as stochastic processes, are not readily available in general purpose languages. Consequently, a non-trivial translation from financial concepts to runnable programs is required. Doing this manually is error prone and time consuming, and it's therefore vital to make financial concepts available either on the library or language level.

The closer the library or language models the notation used by the domain experts, the less manual translation is needed. The converse also holds; it's easier to understand what the program does if it uses a familiar notation. The combination of these two properties allows for quick prototyping.

As a small example of what we want to achieve, it should be straightforward define the Brownian motion and the standard underlying, as defined in the introduction.

Section 4.1.1 and 4.1.2 describe two concepts that are prevalent in quantitative analysis, and as such should be supported by the language.

### 4.1.1 Stochastic processes

By now the need for stochastic processes should be clear; however, some details are still missing.

#### Continuous time

Stochastic processes can be seen as a function from time to a distribution. When using this interpretation, we will write $p(t)$ for looking up the distribution at time $t$ in process $p$. In high level specifications, as opposed to those tied to a specific computational model, the time is usually a (non-negative) real number, and we thus have *continuous-time* stochastic processes [BBG97].

#### Infinite processes

It does not always make sense to talk about the end time of a process. Stochastic processes are thus often specified as being infinite, which consequently should be supported. In the end, we usually want to look up at a specific time, or in a specific time interval, but this decision is best deferred so that the same process can be reused for lookups at different times.

#### Arithmetic operators

The arithmetic operators required by the *CC* pricer are a given. However, *pointwise* perhaps needs some explanation when talking about continuous time processes; what we mean is that for any arithmetic operator $\oplus$, for any two processes $p$ and $q$, at any time $t$, $(p \oplus q)(t) = p(t) \oplus q(t)$.

### 4.1.2 Distributions

Many stochastic processes are specified in terms of distributions, and can be seen as functions from time to a distribution. The properties of the distributions thus directly affect the properties of the stochastic processes.

#### Continuous sample space

Continuous distributions such as $\mathcal{N}$ allows us to more closely match the mathematical specification of distributions and stochastic processes with the specifications written in the language.

#### Dependent distributions

Dependent distributions and stochastic processes are required to compose larger distributions and processes from smaller ones, such as specifying the standard underlying in terms of a Brownian motion.

## 4.2 Composability and reuse

It's important that complex prices can be built from simpler parts, that can be designed, reasoned about, and tested in isolation. This increases confidence in the system and reduces development time.

### 4.2.1 Composable processes for composable pricing

The pricer for composable contracts builds prices for large contracts by composing the prices of smaller contracts. Consequently, the stochastic processes in which prices are expressed must be composable.

### 4.2.2 Discretization as a separate concern

The concern of specifying stochastic processes should be separated from the concern of choosing a computational model, discretization, etc. This allows the components to be reusable across computational models. A price can be defined as a stochastic process once, and then be queried using the best computational models available for that process and query.

## 4.3 Supporting a wide range of contract prices

If only the price of a few financial contracts are expressible in the language, there is little advantage over writing an ad-hoc pricer for each. There is work involved in creating a language for at least two parties; the language designers must design and implement the language, and the domain experts must learn the language and implement pricers. The language approach is thus only worthwhile when the work involved in specifying ad-hoc pricers for each financial contract outweighs the cost of creating the language. The more prices that can be expressed in the language and the easier it is to learn and use, the greater the chance that this balance will tip in favour of creating the language. It is thus important that we can express the price a wide range of financial contracts.

### 4.3.1 Conditionals

Conditionals was implicitly required to implement the *CC* pricer. Additionally, pricing certain options such as barrier options requires conditional logic. We thus need a way to express this logic.

### 4.3.2 Multiple sources of uncertainty

Financial contracts like basket options may depend on multiple underlyings, each of which is likely to contribute its own source of uncertainty. *CC*

supports these and we thus need to be able to specify prices in terms of multiple sources of uncertainty.

### 4.3.3 Forecasting

In order to price American and Bermuda style options, we need to determine whether or not to exercise the option at any given time. In order to do this, we need a way to query the expected value of the stochastic process in the future.

### 4.3.4 Aggregation

As we have already seen in the introduction, pricing Asian options requires taking the average over a process in a time interval. Lookback options and other *path dependent* prices are similar, but require other aggregations than the average. We thus need a sufficiently general aggregation construct to support these financial contracts.

## 4.4 Having clear semantics

In order to properly reason about what a program in any language does, formal semantics is required. In this case, there are already a range of probabilistic languages to borrow from, whose semantics are clearly defined.

## 4.5 Yielding efficient implementations

Management of large portfolios of financial contracts and high frequency trading both require efficient pricing to be feasible. Additionally, it's convenient when developing pricing code, that the price can be obtained almost instantly, both for prototyping and running automated tests. If the language is not designed with this in mind, it might not be possible to build an efficient implementation in practice.

# Chapter 5

# Probabilistic functional programming

We have seen that the pricing of financial contracts requires a great deal of probabilistic programming. The *CC* library presumed the existence of a data type for stochastic processes in its implementation of the pricer and it assumes that every observable is convertible to a stochastic process. Probabilistic calculations are also required to implement the discounting and currency exchange in the model as this again needs to be done on stochastic processes.

This is all needed to price *CC* contracts: a probabilistic programming library providing stochastic processes as well as functionality to find the expected value of these processes at certain times. The library would also need to be powerful enough to implement the desired observables and financial model. In the end, we would like to find the expected value efficiently, and our approach will be to calculate it on the GP-GPU. This means that we need to eventually translate the probabilistic programs into GP-GPU code, and in turn this means that the bulk of the computation required to find the expected value must be symbolic.

Our initial plan was to find a suitable Haskell library or embedded language for probabilistic functional programming, give it a back end that could simulate expected values using GP-GPUs, and use it to price *CC* contract. As it turned out, we did not find any existing library suitable and ended up building our own probabilistic language, namely *SPL*. Nevertheless, this chapter provides insight into how these libraries or languages function in general, point out their differences and compare them to our specific needs. We will later use concepts from this chapter to explain *SPL*.

## 5.1 Discrete distributions

Erwig and Kollmansberger describes in [EK06a] a probabilistic functional programming library for Haskell. The basic idea is to represent a distribution as a list of all possible outcomes (the *sample space*) coupled with their probability, which is a real number between 0 (impossible) and 1 (certain):

```
data Dist a = D [(a, Probability)]
```

When using the library, distributions are not constructed directly via the `D` data constructor, but are instead constructed via a set of provided functions. For example, given a list of values, `uniform` constructs a *discrete uniform distribution*; that is, a distribution with a finite number of equally probable values. We might thus define flips of a balanced coin as:

```
data Coin = Heads | Tails

flip :: Dist Coin
flip = uniform [Heads, Tails]
```

Printing this distribution would allow us to verify probabilities:

```
> flip
Heads 50%
Tails 50%
```

Independent distributions can be combined via the `joinWith` function. For example, we might want to know what the distribution of flipping two coins is:

```
both a b = [a, b]

flip2 = joinWith both flip flip
```

Again, we can view the probability of each combination by printing it:

```
> flip2
[Heads, Heads] 25%
[Heads, Tails] 25%
[Tails, Heads] 25%
[Tails, Tails] 25%
```

We might like to know how often `Tails` occurs at least once during two flips of a coin. The library provides an implementation of `Functor Dist`, so we can use `fmap` for this[1]:

[1] Since this is a predicate, you may prefer: `flip2 ?? any (== Tails)`.

```
> fmap (any (== Tails)) flip2
False 25%
 True 75%
```

Note that the list representation has many equivalent ways of representing distributions. In fact, the distribution above is internally stored as `D [(False, 0.25), (True, 0.25), (True, 0.25), (True, 0.25)]`. The pretty printer performs *normalization*, which collates equivalent values by summing their probabilities. If the sample space is small, normalization can prevent the representation from exploding.

We may design a game of tossing coins as follows: The player tosses a coin. If the coin comes out heads, he gains a point and then tosses the coin again; if the coin comes out tails, he gets no more points and his turn is over.

However, whether or not we toss another coin is now *dependent* on the previous coin toss. This dependency is expressed via the monadic bind operator `>>=`. To complete the monad, `return a` constructs the distribution consisting only of a single value, `a`, with probability 1; a so called *degenerate distribution*. With this, we can model the distribution of scores in our game:

```
toss :: Integer -> Dist Integer
toss 0 = return 0
toss n = do
    coin <- flip
    case coin of
        Heads -> do
            score <- toss (n - 1)
            return (score + 1)
        Tail -> return 0
```

The question is then, what are the probabilities of getting each score? To make the game finite, we limit the maximum number of tosses:

```
> toss 5
0 50%
1 25%
2 12.5%
3 6.25%
4 3.125%
5 3.125%
```

We might make the additional rule that people with a history of bad luck gets to retry indefinitely whenever their score is even. Instead of altering the definition of `toss`, we can add a guard[2]:

[2]At least in principle; we have not run this last example, because the implementation we've used here does not seem to handle guards correctly.

```
> do score <- toss 5; guard (odd score)
1 50%
3 25%
5 25%
```

Internally, `guard False` generates the `impossible` distribution represented by `D []`. Guards are essentially filters on the sample space, and as with filters, a narrow predicate will remove most of the results generated so far. Since the results must be generated before they can be filtered, guards are often more costly than simply generating a smaller distribution from the beginning. On the other hand, guards are sometimes more convenient.

## 5.2 Symbolic representation

With the list representation, every `joinWith` and `>>=` creates the Cartesian product of the distributions. If the sample space grows similarly, normalization is not enough to prevent the exponential growth resulting from a linear number of applications of these operators. Keeping a large list in memory is at best detrimental to performance and at worst simply not possible. In an effort to reduce the memory usage, a symbolic representation was presented in [Lar11].

The representation of distributions is very different from what we have seen so far:

```
data Dist a where
    Certainly :: a -> Dist a
    Choice    :: Probability -> Dist a -> Dist a -> Dist a
    Fmap      :: (a -> b) -> Dist a -> Dist b
    Join      :: Dist (Dist a) -> Dist a
```

This example uses GADTs, which we will return to later, but for now the important thing to notice is that the internal representation is a data structure containing the *operations* required to compute the distribution, and not a flattening of the distribution itself. Often the *expected value* (mean weighted by probability) of the distribution is the query itself. Instead of flattening the distribution and then computing the expected value, this representation allows the lazy unfolding of the distribution, keeping the memory usage constant. The reduced memory usage leads to a large constant speed-up, although the asymptotic computation time is unchanged.

The distribution still forms a monad, and it's straightforward to express `uniform`, so we can run most of the previous examples unchanged. Guards are missing from this definition, but could likely be added easily.

Note that not all computational steps are kept symbolic in this representation. In particular, the `a -> b` in `Fmap` is an arbitrary Haskell function and is thus beyond inspection (and therefore optimization) by the library.

## 5.3 Stochastic processes

Probabilistic functional programming has been applied to counter-intuitive problems that map directly to distributions, such as the Monty Hall Problem [EK06a] and the Flu Test False Positive Problem [Kid07]. However, it has also been used to model distributions that change over time, such as tree growth [EK06a], predator/pray populations and genome evolution [EK06b]. These are called *stochastic processes*.

The approach taken by Erwig and Kollmansberger is to model stochastic processes as a list of distributions, `[Dist a]`. A transition function `a -> Dist a` is then iterated a number of times, producing a list of the distribution at each time step. For example, we may model the price of a fictive asset that goes up or down by 1 with equally probability at each time step as follows:

```
price :: Double -> Dist Double
price s = uniform [s - 1, s + 1]
```

We can observe the evolution of the price by iterating the `price` transition function using the built-in operator `*.`. The example below shows the possible ways for the `price` process to evolve from time 0 to time 3, starting at value 5:

```
> (3 *. price) 5
[5 100%,

 4 50%
 6 50%,

 3 25%
 5 50%
 7 25%,

 2 12.5%
 4 37.5%
 6 37.5%
 8 12.5%]
```

Although the output is normalized, the internal representation may not be. In that case, the size of the representation grows exponential with the number of steps we take, as visualized in figure 5.1.

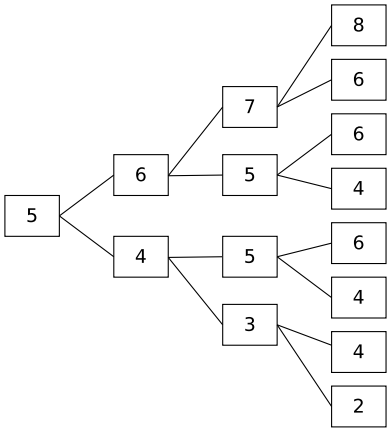



Figure 5.1: Internal non-normalized representation for each step of `(3 *. price) 5`. The probabilities are represented by the number of paths leading to each sample divided by the total number of paths leading to that depth.

The opportunities for normalization is visualized on the left in figure 5.2. In this case we're fortunate that the sample space only grows linearly, which leads to a linear growth in a normalized representation as visualized on the right.

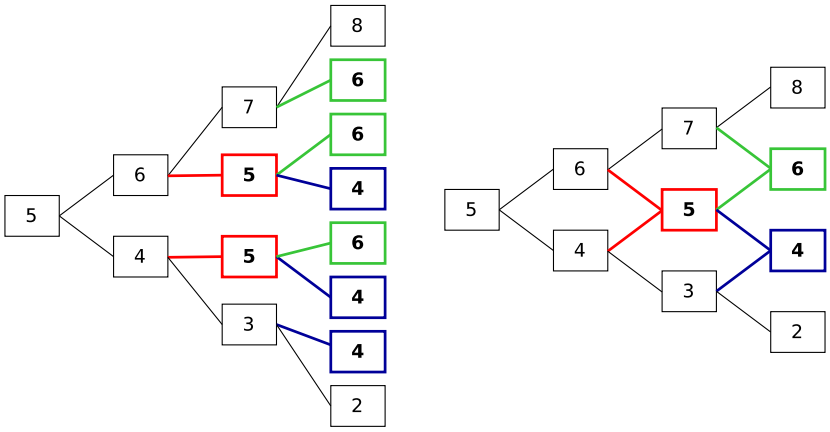



Figure 5.2: The opportunities for normalization (on the left, in bold) and the internal normalized representation (on the right) for each step of `(3 *. price) 5`.

This specific example yields a *recombining binomial tree* [JE03], since every node has two branches, and going down and then up or up and then

down will lead to the same node. When applicable, this is an efficient representation.

In this example and the examples mentioned, the time step is implied in the definition of the processes. Consequently, two processes are only combinable, if defined using the same time step. The list representation of processes are therefore non-composable when the time step is not a universal constant. A composable process definition, still using discrete time, with time steps, will therefore require processes to be defined parametrically according to the time step.

The libraries mentioned in this chapter only provide constructors for discrete distributions. However, some stochastic processes are specified in terms of *continuous distributions*. An example of this is the *Brownian motion*[3] $\mathcal{W}$, which is specified in terms of the continuous standard normal distribution $\mathcal{N}$ and may be formulated like this:

$$\begin{aligned} \mathcal{W}_0 &= 0 \\ \mathcal{W}_{t+\Delta t} &= \mathcal{W}_t + \mathcal{N}\sqrt{\Delta t} \end{aligned}$$

Although $\mathcal{N}$ can be approximated discretely via a series of equally likely yes/no questions (a *binomial distribution*), the sample space, being an approximation of the real numbers, quickly grows too large to store in a list.

As we have already seen, many financial pricing problems are specified in terms of the Brownian motion and are themselves continuous stochastic processes. As such, it would be natural to offer continuous distributions and stochastic processes as the building blocks of a language for the financial domain.

## 5.4 Monte Carlo simulation

Instead of constructing a list representing the exact distribution, we can use a pseudo random number generator (*PRNG*) to pick random samples from the distribution, and thus avoiding the computation of the entire distribution. This approach is called Monte Carlo simulation, and is often used in pricing financial contracts [Hul09].

The idea is that every time we find a primitive distribution, we pick one of the samples from the sample space, at a chance corresponding to its probability. The law of large numbers says that as we increase the number of random samples, the mean of the random samples approaches the expected value of the distribution.

As an example, consider the asset price example from section 5.3. We may select a subset of the time series at random via Monte Carlo simulation, as visualized in figure 5.3.

[3] Sometimes called a Wiener process.

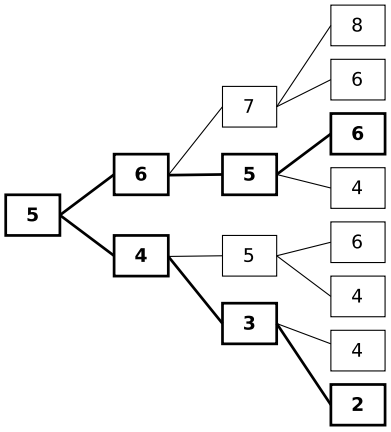



Figure 5.3: Randomly selected subset (in bold) of all possible time series.

[Kid07] uses monad transformers to build a probability monad similar to the one in which we did coin flipping. One of these transformers enable Monte Carlo simulation for any distribution defined via the other monad transformers, separating the concerns of specifying the distribution and querying it. Consequently, the distribution of stochastic process can be specified once and then later computed exactly or sampled via Monte Carlo simulation.

## 5.5 Summary

The libraries covered above does not allow us to extract the calculations involved in building distributions. This makes us unable to translate the computations on distributions into GP-GPU code for doing Monte Carlo simulation. [EK06a, EK06b, Kid07] do not use a symbolic representation at all, and it would not be possible to make them do so while preserving the monadic bind. [Lar11] is not symbolic in its argument to `fmap` which takes an arbitrary Haskell function of type `a -> b`. This shows that both `Monad` as well as the weaker class `Functor` is not viable instantiations for the `Dist` data type when `Dist` needs to be fully symbolic.

All of the libraries use discrete distributions. This is not a problem in general, but as we have seen in section 4, continuous distributions like $\mathcal{N}$ are needed, and having to decide on a discretization early is problematic. Representing stochastic processes as lists of distributions is also not ideal in our case, because this requires early discretization of the time step.

We will instead design a separate embedded probabilistic language that attempts to fulfil all of the requirements specified in chapter 4.

We will present this language in chapter 7, just after the next chapter where we present two embedded languages for performing calculations on the GP-GPU directly from Haskell.

# Chapter 6

# Array languages targeting GP-GPUs

In an effort to provide a high level interface to programming for GP-GPUs, several embedded array languages for Haskell have been proposed. Amongst these are Nikola [MM10] and Accelerate [CKL+11], which will be discussed in this section.

GP-GPUs provide a massively parallel computational model with shared memory, where every thread runs the same code, but with different parameters. As such, it's easy to see how a function like `map` – which applies the same function to every element – fits into this model. However, even aggregating functions like `fold` and `scan` can be efficiently implemented [CBZ90]. We will cover GP-GPUs in more detail in section 8.5.

Since GP-GPUs have their own memory separate from that accessible from the CPU, it's necessary to have a strategy for copying arrays from the host to the device and back. Nikola does this both ways automatically whenever an array expression is evaluated, while Accelerate makes the memory region of an array explicit in the type system and provides instructions for copying.

Nikola and Accelerate are both *deeply embedded* into Haskell, meaning that while they provide Haskell functions and types on the surface, they internally capture and store the computational steps in a data structure.

The idea is that instead of having the library functions directly compute the result array, they instead compute a syntax tree that can then later be transformed, translated and evaluated.

Haskell makes this very convenient by making it possible to instantiate the `Num` type class for custom types, providing overloads for common operators like `+`, `-` and `*`. Syntactically, code that only uses the operators from this type class (or any of the other suitable type classes) is the same whether they work on `Integer`, `Double` or a custom type like `Exp Float` in Nikola.

Functions in both languages are represented directly by Haskell func-

tions. This approach is called *higher order abstract syntax*, or HOAS. We will return to it in chapter 8.

However, the sharing implied by Haskell bindings is not readily observable in the computed syntax tree. In effect, everything is inlined where it's used. In order to prevent recomputation and code size growth due to this, Nikola discovers sharing via *stable names* [Gil09, SME99] in the IO monad, which allows reification of let bindings. In order to capture function application syntactically, a special version of `$` is supplied.

Note that we can only use a deeply embedded language if we use the operators it provides. In particular, if we wanted to use Nikola or Accelerate to speed up the probabilistic libraries discussed in chapter 5, we would be stuck with functions internally using Haskell's plain operators for scalar types, like `+`, `-` and `*`, and not those of Nikola or Accelerate. We would thus need to maintain the deep embedding all the way up to the surface of the probabilistic language.

While our initial plan was to pursue this option, we could not get the tests of either library to run on our hardware. Geoffrey Mainland kindly fixed this issue for Nikola on our system, but by then we were working on a different approach. Whether or not the language we propose can be implemented in terms of Nikola or Accelerate is thus postponed as future work.

However, our approach to embedding is heavily inspired by that of these languages. We will return to this in chapter 8.

# Chapter 7

# A stochastic process language - SPL

We will now return to the requirements identified in chapter 4 and attempt to specify a language that fulfils them.

We have already seen languages that each solved a subproblem of what we want to achieve. While it's tempting to conclude that we can simply combine these languages to provide a complete solution – indeed, this was our initial plan – it isn't quite as straightforward as one could hope for. The languages for probabilistic functional programming, we have seen so far, do not make all the computational steps inspectable; thus it is not possible to translate these computational steps into operations in the array languages for the GP-GPU.

Part of the ingenuity of the array languages visited in chapter 6 is their approach to embedding into Haskell. However, to provide a higher level language for distributions and stochastic processes, the embedding must be redone for this higher level language. In any case, our approach to embedding will be heavily inspired by these array languages.

In order to do Monte Carlo simulation, it would be necessary to find or implement a pseudo random number generator that supports the array languages, since this is likely a significant part of the computation involved.

The benefit of using an array language, in this context, would then primarily be the generation of GP-GPU code. Since this in itself is a significant part of the implementation, it could still be a good idea to use them. However, the implementation (and thus optimization) of the higher level language would need to be expressed in terms of array operations, which imposes some restrictions compared to the lower level C code.

Combined with not being able to run the array languages for the first part of the project, this prompted us to simply build and implement a separate language, reusing ideas from, but not the implementation of, these languages.

## 7.1 Language design

In this section we present the language design of SPL, as motivated by chapter 4.

### 7.1.1 Built-in constructs

As mentioned in section 4.1.2, we need distributions with continuous sample space. For this purpose, we assume the existence of the type of real numbers, called `Real`. We can then specify some built-in continuous distributions:

```
uniform, normal :: Dist Real
```

The standard normal distribution is provided due to its prevalence in finance, such as in the definition of the Brownian motion. The standard uniform distribution is provided because there are many implementations of pseudo random number generators available that sample from a uniform distribution, and as such it is likely the most efficient basis from which to build other distributions[1].

As mentioned in section 4.1.2, we need to be able to express that one distribution *depends* on another, for which we have the following construct:

```
sample :: Dist a -> (Dist a -> Dist b) -> Dist b
```

This is similar to the monadic bind used to serve the same purpose in [EK06a]. However, the monadic bind would have type `Dist a -> (a -> Dist b) -> Dist b` and thus require that the given function took something of type `a`, which we wouldn't be able to reify; hence the type above.

As mentioned in section 4.1.1, arithmetic operators are required to implement the pricer. For `Dist Real` we provide instances of the standard type classes `Num`, `Fractional` and `Floating`[2]. This gives us arithmetic operators like `+`, `-`, `*`, `/` etc. as well as `pi`, `exp`, etc. This allows us to treat `Dist Real` as a Haskell built in number type, save for pattern matching.

In order to support *CC* and barrier options, we need conditional logic as mentioned in section 4.3.1. We would prefer to implement the `Ord` type class, but unfortunately the operators return a naked `Bool`. Instead, we provide the following type class and an instance `Ordered Dist Real`, with semantics and operator precedence corresponding to that of the `Ord` type class:

[1] Indeed, we use it in our implementation of `normal`, see section 8.

[2] The naming of the `Floating` type class is a bit unfortunate, because although we will use a floating point discretization of the real numbers, there's nothing inherently floating point about providing $\pi$ etc.

```
class Ordered a b where
    min_,
    max_    :: a b -> a b -> a b
    (.<.),
    (.<=.),
    (.>.),
    (.>=.),
    (.==.),
    (./=.)  :: a b -> a b -> a Bool
```

For `Dist Bool` we provide the below type classes and instances for `If_ Dist` and `Boolean Dist`. These correspond to the if statement and the usual boolean operators:

```
class If_ a where
    if_     :: a Bool -> a b -> a b -> a b

class Boolean a where
    (.||.)  :: a Bool -> a Bool -> a Bool
    (.&&.)  :: a Bool -> a Bool -> a Bool
    not_    :: a Bool -> a Bool
```

Additionally, we provide pair types via the following type class and an instance for `Pair Dist`:

```
class Pair a where
    pair    :: a b -> a c -> a (b, c)
    first   :: a (b, c) -> a b
    second  :: a (b, c) -> a c
```

Constants of type `Double` and `Bool` can be lifted to distributions via an instance of the following type class:

```
class ToConstant a where
    type ConstantType a
    constant :: ConstantType a -> a
```

In SPL, time is modelled as a non-negative real number, yielding continuous-time processes as mentioned in section 4.1.1. A stochastic process is conceptually a function from a time to a distribution, and this concept is realized in the following operator, which takes a time and a process and returns the distribution of the process at that time:

```
lookup :: Dist Time -> Process a -> Dist a
```

Note that the time can be determined at runtime, as is evident from the type of the time parameter.

The same interpretation of processes can be used to construct them:

```
closed :: (Dist Time -> Dist a) -> Process a
```

Note that no end time is specified; this is an infinite process as specified in section 4.1.1.

As mentioned in section 4.3.4, in order to support path dependent processes we need to aggregate over time intervals. We provide one part of this functionality via the following construct which defines a process from another process by accumulating over it. The following operator is somewhat similar to the to `scanl` in Haskell[3], except that operator also takes the time difference between the previous process value and the current:

```
prefix ::
    (Dist Time -> Dist a -> Dist b -> Dist a) ->
    Dist a -> Process b -> Process a
```

Note that these processes are also infinite. The other part of aggregating over a time interval is provided by the following function that skips ahead in a process by a specified amount of time:

```
skip :: Dist Time -> Process a -> Process a
```

Processes may run *in parallel* via the following operator, which is analogous to Haskell's `zip`:

```
zip :: Process a -> Process b -> Process (a, b)
```

In order to express that one process *depends* on another, we have the following construct:

```
trace :: Process a -> (Process a -> Process b) -> Process b
```

Again, the type seems similar to the monadic bind, but isn't for the same reason that the type of `sample` isn't.

We shall later see that the internal representation is fully symbolic, and also how to separately specify the global time step. This concludes the built-in constructs of the language. The next section will focus on how common functionality can be expressed in terms of these.

[3]The `prefix` name is derived from its similarity to the *exclusive prefix sum*.

### 7.1.2 Prelude functions

We can define the process whose value is the current time by passing the identity function to `closed`:

```
time :: Process Time
time = closed id
```

We may define the weighted binary choice and discrete uniform distributions from the constructs we have already seen as follows:

```
choice :: Dist Real -> Dist a -> Dist a -> Dist a
choice q d1 d2 = if_ (uniform .<. q) d1 d2

choose :: [Dist a] -> Dist a
choose [d] = d
choose (d:ds) = choice uniformly d (choose ds)
    where
        uniformly = 1 / fromIntegral (length (d:ds))
```

The `prefix` construct can be used to express `map` for processes by ignoring the delta time and accumulator and simply applying the given function to the value of the process at a given point. The initial value needs not be defined since it's never used:

```
map :: (Dist a -> Dist b) -> Process a -> Process b
map f = prefix (\_ _ d -> f d) undefined
```

Having this we can apply unary `Dist a` operators to `Process a`. Since we would also like to apply binary and ternary operators, we define the lifting functions for `Process a` as specified by [JES00]. Before we do this, it is convenient to specify the equivalent of `curry` and `uncurry` in Haskell:

```
uncurry   f v      =  f (first v) (second v)
curry     f a b    =  f (pair a b)
```

We right-iterate pairs to emulate n-tuples:

```
uncurry3  f v      =  uncurry (f (first v)) (second v)
curry3    f a b c  =  f (pair a (pair b c))
zip3        a b c  =  zip a (zip b c)
```

And finally it's straightforward to define the lifting functions:

```
lift               =  map
lift2  f p1 p2     =  map (uncurry  f) (zip  p1 p2)
lift3  f p1 p2 p3  =  map (uncurry3 f) (zip3 p1 p2 p3)
```

Using these, we implement the same type classes for `Process Real` and `Process Bool` as we have for `Dist Real` and `Dist Bool`, meaning that operators such as `+, -, *, ...` are *pointwise* operations on processes.

Since `prefix` is exclusive of the initial value, we might also like a variant that is inclusive. The definition here is a bit tricky, since we use a pair to delay the output by one time step in order to push in the initial value:

```
inclusivePrefix f v p = map first (prefix f' (pair v v) p)
    where
        f' dt a v = pair (second a) (f dt (second a) v)
```

We specified the Brownian motion as an iterative process. In particular, it is a function that is iterated over an initial value with the $\Delta t$. We therefore define the `iterative` construct:

```
iterative :: (Dist Time -> Dist a -> Dist a) -> Dist a -> Process a
iterative f i = inclusivePrefix (\dt a _ -> f dt a) i time
```

Note that we don't care about the process argument, but the time process is as good as any. Recall the definition of the Brownian motion:

$$\begin{aligned} \mathcal{W}_0 &= 0 \\ \mathcal{W}_{t+\Delta t} &= \mathcal{W}_t + \mathcal{N}\sqrt{\Delta t} \end{aligned}$$

This is now straightforward to write down:

```
brownian :: Process Real
brownian = iterative (\dt w -> w + normal * sqrt dt) 0
```

The `prefix` construct also allows us to aggregate over a stochastic process. We can define the cumulative moving average of a process by dividing the sum of the values seen so far by the number of time steps taken so far:

```
average process = process `trace` \p -> total p / count p
    where
        total = prefix (\_ a v -> a + v) 0
        count = prefix (\_ a _ -> a + 1) 0
```

Note that we use `trace` here to ensure that `total` and `count` are looking at the same time series of the process.

The Brownian motion and the average are components of the Asian option price, but we have now reached the border between common functionality that belongs in the prelude and the specifics of the Asian option price. We shall return to the pricing of these options later.

### Representation concerns

In the above we have ignored a detail, namely the requirement that all type parameters to `Dist` and `Process` must be instances of our `Type`. This constraint ensures that only types representable on the target platforms are used. The instances provided are `Type Real`, `Type Bool` and `(Type a, Type b) => Type (a, b)`:

```
class Typeable a => Type a where
    splType :: a -> SPLType
    toDist  :: a -> Dist a
```

The `splType` is analogous to Haskell's `Data.Typeable.typeOf`, and `toDist` ensures that we can convert any value of `Type a => a` into `Dist a`. The `Typeable a` constraint is used in the conversion to intermediate code, which will be covered in chapter 8.

For the sake of readability, we will leave these constraints out of this document, but they are present in the full code listing in appendix B.

### 7.1.3 Haskell's bindings vs. `sample` and `trace`

While `sample` and `trace` ensures that all occurrences of the bound variable refer to the *same* sample or time series during any evaluation of the body, using Haskell's bindings allows the occurrences to be *different* samples or time series. The latter semantics is required; otherwise we could not reuse eg. `brownian` to introduce multiple sources of uncertainty. When the bound value is only used in one place, the difference in semantics is not observable, which makes it possible to use Haskell's more convenient bindings quite often.

### 7.1.4 Semantics

Given a finite uniform discretization of the time:

```
[0, delta .. end]
```

We can give semantics to SPL in terms of any probability monad that can provide the following distributions:

```
class Monad m => ProbabilityMonad m where
    uniform' :: m Real
    normal'  :: m Real
```

Note that all of the libraries we have seen in chapter 5 can provide discretized versions of these. Our representation actually requires that `Real`

is discretized as `Double`, although this is an implementation artefact and not an inherent limitation.

We're going to use the abstract syntax representation given in section 8.2, but it's a direct representation of the built in constructs, and the names are the same modulo capitalization except for `Certain`, `Unary`, `Binary` and `Ternary` that provide constants, unary, binary and ternary operators respectively. We can now give the semantics of stochastic processes that defines them as distributions of time series:

```
process :: ProbabilityMonad m => Process a -> m [a]
process p = case p of
    Closed f ->
        mapM (distribution . f . Certain . Double) [0, delta .. end]
    Prefix f i p | usesAccumulator f -> do
        i' <- distribution i
        p' <- process p
        let accumulate a v =
                distribution (f (toDist delta) (toDist a) (toDist v))
        l <- scanM accumulate i' p'
        return (tail l)
    Prefix f i p -> do
        p' <- process p
        mapM (distribution . f (toDist delta) undefined . toDist) p'
    Zip p1 p2 -> do
        p1' <- process p1
        p2' <- process p2
        return (zip p1' p2')
    Trace p f -> do
        p' <- process p
        let s = Closed (\(Certain (Double t')) -> toDist (index t' p'))
        process (f s)
```

The case for `Prefix` examines the function to see if it uses the accumulator argument. This is only required due to the use of `undefined` in the definition of `map`, and is possible because our representation is completely inspectable. The `usesAccumulator` function is defined in module `Language.SPL.Syntax`. Other than that, it is either an application of `scanM` or `mapM`. The former function is not part of the standard Haskell prelude, but is straightforward to define for all monads, which is done in module `Language.SPL.Semantics`.

The semantics would be simpler if `map` was simply built in; then you could regard the first case as the semantics for `prefix` and the second case as the semantics for `map`, thus removing the need for `usesAccumulator`.

The case for `Trace` uses a `Closed` process to lift a time series into a process. Note that the pattern match on `Certain (Double t')` works pre-

cisely because the case for `Closed` calls `f` with exactly that structure, and `f` is never called elsewhere. The `index` function should simply look up into the time series at the specified time; however, our internal representation of time (namely `Double`) unfortunately leaks here, and incurs a little bit of noise to convert it into an integer that can be used as an index into the list:

```
index t l = l !! floor (t / delta)
```

The semantics for distributions are straightforward. Note however that `unaryOperator`, `binaryOperator` and `ternaryOperator` simply convert the symbolic operators into the corresponding operators from the Haskell prelude. Note that all operators are strict, including the if statement:

```
distribution :: ProbabilityMonad m => Dist a -> m a
distribution d = case d of
    Uniform -> uniform'
    Normal -> normal'
    Certain (Double v) -> return v
    Certain (Bool v) -> return v
    Lookup t p -> do
        t' <- distribution t
        p' <- process p
        return (index t' p')
    Sample d f -> do
        d' <- distribution d
        distribution (f (toDist d'))
    Unary o d -> do
        d' <- distribution d
        return (unaryOperator o d')
    Binary o d1 d2 -> do
        d1' <- distribution d1
        d2' <- distribution d2
        return (binaryOperator o d1' d2')
    Ternary o d1 d2 d3 -> do
        d1' <- distribution d1
        d2' <- distribution d2
        d3' <- distribution d3
        return (ternaryOperator o d1' d2' d3')
```

The semantics for `skip` are given as a transformation on the syntax tree in section 8.2.

## 7.2 Implementing a *CC* model

Part of the initial purpose of *SPL* was to provide a language well suitable for implementing *CC* models. As described in section 3.2, this involves implementing the three functions *exch*, *disc* and *snell*. We group these functions together to form a `Model` type in Haskell

```
data Model = Model {
    modelExchange :: Currency -> Currency -> Process Real,
    modelDiscount :: Currency -> Contract -> Time -> Process Real,
    modelSnell :: Currency -> Contract -> Time -> Process Real}
```

Recall that the abstract pricer $\mathcal{E}[\![]\!]$ finds the price process of a contract given a model and a currency. As we are implementing this using the *SPL* `Process` type as the *CC* stochastic process data type, we obtain a pricer having the signature

```
price :: Model -> Currency -> Contract -> Process Real
```

Now let us defined a simple financial *CC* model. This is an easy task when first provided the three essential functions defined further below.

```
model :: Model
model = Model exchange discount snell
```

`exchange` defines the time varying exchange rate between two currencies. This function must obey two properties given in [JES00]

$$\begin{aligned} exch_k(k) &= \mathcal{K}(1) \\ exch_{k2}(k_1) \cdot exch_{k3}(k_2) &= exch_{k3}(k_1) \end{aligned}$$

and as a consequence of this also

$$exch_{k2}(k_1) \cdot exch_{k1}(k_2) = exch_{k1}(k_1) = \mathcal{K}(1)$$

As we only support the currencies DKK and USD, this only leaves us one process to implement

```
exchange :: Currency -> Currency -> Process Real
exchange a b | a == b = 1
exchange DKK USD = max_ 0 (5.268 + brownian * 0.1)
exchange USD DKK = 1 / exchange DKK USD
```

The rate for exchanging DKK to USD is starting from 5.268, but is from then on given some fluctuation increasing over time modelled by the Brownian motion. The rate will never drop below zero. We have simple taken this exchange rate out of the thin air. In a real setting, this model should be implemented by a financial expert.

We implement a discounting model assuming a continuous fixed risk-free interest rate of 5%.

```
discount :: Currency -> Contract -> Time -> Process Real
discount currency contract t =
    let p = price model currency contract in
    let discounter = exp (-0.05 * (constant t - time)) in
    let discounted = always (lookup (constant t) p) * discounter in
    if_ (time .<=. constant t) discounted 0
```

The discounting function first finds the non-discounted price process `p` using the pricer together with the model we are about to define. It then calculates the discounting factor process, `discounter`, discounted from time `t`. This factor is then multiplied on the process always yielding the future pay out value at time `t`. We have ensured that the price after `t` is zero even though this is undefined in the abstract semantics. Note that the type class function `constant` is used in two different settings. It is first used to convert the time `t` to a constant process and later to convert the time to a distribution.

We still need a definition of `snell` as required by the model and this will be the topic of the following subsection.

### 7.2.1 Decisions based on the (expected) future

The snell envelope implementation given below is *not* correct. We bring it here anyway, to illustrate the problem of pricing contracts with multiple exercise times, when using Monte Carlo simulation.

```
snell :: Currency -> Contract -> Time -> Process Real
snell currency contract end =
    let p = price model currency contract in
    let discounted t = exp (-0.05 * (constant end - always t)) * p in
    let envelope t = maximum_ t (constant end) (discounted t) in
    if_ (time .<=. constant end) (closed envelope) 0
```

As in `discount`, we first calculate the price process `p` of the contract. We then construct the function `envelope` that given a potential exercise time `t`, finds the maximum price from `t` to `end` after discounting the value back to time `t`. Finally this function is converted into a process using the `closed` constructor.

The intuition behind this implementation is, that the contract holder will not exercise the contract at time `t`, if there exists a later exercise time that would yield a larger profit, even after discounting that value back to time `t`. The problem behind this intuitions is, that this is only true, if the contract holder can see into the future. We assume the holder to be completely rational but not psychic. The right intuition is therefore better stated as: The holder will not exercise the contract at time `t` if the holder *expects* a later exercise time that would yield a larger profit.

The rational holder could instead try to estimate the future using simulations, as we do. An approach to calculating the right price is therefore to do the same i.e. do simulations in our simulation. This is achieved by replacing `p` in the definition of `discounted t` by a new process, say `expectedP t`, simulating the possible futures starting with the value of `p` at time `t`.

It is unfortunately not possible to express nested simulation directly in *SPL*. But as *SPL* is a embedded language, it is possible to generate or unroll an *SPL* program that does a fixed number of nested simulations. The problem is just, that the program size will be proportional to $splits^{exerciseTimes}$ where *splits* is the number of nested iterations and $exerciseTimes$ is the number of exercise times in the exercise period. We do therefore not only see an exponential run time but also an exponential code size. The exponent $exerciseTimes$ would usually be $(expiryTime - acquisitionTime)/deltaTime$ which should consolidate the point, that this is not a practical solution.

We thereby have to conclude that *SPL* is not a suitable language for implementing the *CC* model function *snell*. This is unfortunate as *SPL* was actually designed to implement *CC*. But the problem gets even worse as our limitation is not limited to *snell*. As we do the Monte Carlo simulation in time steps we effectively reduce the exercise period of an American option to a finite number of exercise times thereby reducing the American option to a Bermudan options. In itself, this is not a problem as the simulation is an approximation after all. But it made us realize that the psychic problems do not only originate from the use of the *CC* combinator `anytime`. The Bermudan option are also expressible in *CC* as shown below:

```
get (truncate 0 (c `or` (
    get (truncate 1 (c `or` (
        get (truncate 2 (c `or` zero))))))))
```

The missing implementation of *snell* could be handled by taking `anytime` out of the *CC* language, but it is far worse that we cannot handle special combinations of `get`, `truncate` and `or`, as this makes the supported subset of the *CC* language non-composable. The *CC* programs that we do not support, are those expressing contracts where the holder needs to base a choice upon a expected future values. This is the case in the Bermudan op-

tion given above, but let us look at the symbols contract having this problem

`c1` ‘`or`‘ (`get truncate t c2`)

This contract lets the holder choose whether to take `c1` or the contract that yields `c2` at time $t$. The rational choice is here to pick the option that has the highest expected price. Using simulation, this can be determined by pricing `c1` and `get truncate t c2` in two different simulations and then compare the expected prices. Our current implementation performs one combined simulation where each simulation experiment will yield the price for the choice that has the highest price within the experiment. This is again an exploitation of future knowledge which results in upper bound prices compared to the right prices.

We leave this problem to future work, but we do have an idea that might lead such work in the right direction. What if we say that the current *CC* to *SPL* pricer implemented is perfectly fine, and instead point our fingers at the *SPL* semantics. Is it arguably never correct to utilize future value when modelling a stochastic process. We thereby need to change the semantics to use expected values whenever a future value is requested by `lookup`. This is especially the case when looking up future values on traced processes as they represent a specific reality, where one should be certain about the present and the past, but not the future.

Based on the argument above we might say, that the problems described in this section is simply a product of bad semantics for *SPL* and that the semantics should be fixed such that future lookups in traces would yield expected values. Implementing such a changed semantics with Monte Carlo simulation would require nested simulations or preferably something more clever. There have been several solutions for doing this for American options [LS01, Hul09], but they might be hard to implement efficiently on a GP-GPU as the simulation experiments are no longer independent in these approaches. Another challenge is to construct the process representing the uncertain future starting from the present value of the trace, which is needed to calculate the expected future.

# Chapter 8

# Implementation

We have already seen the interface for the built-in constructs and the prelude functions of SPL. This chapter explains how we translate these constructs via intermediate representations so that they can be run on GP-GPUs in the end. Figure 8.1 gives an overview of the relations between SPL, the intermediate representations and the execution targets.

Note that we will not attempt to recover sharing as in Nikola, in part due to the semantics introduced in section 7.1.3, which requires re-evaluation of at least the random parts of stochastic terms not explicitly `sample`d or `trace`d. It would be possible to recover the sharing and produce C functions rather than using C's strict bindings, but this approach will be left to future work.

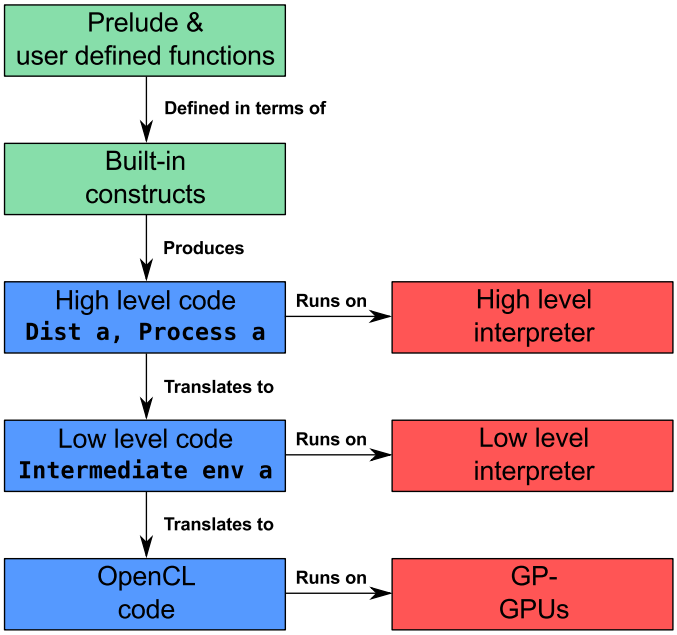



Figure 8.1: SPL interfaces, internal representations and targets. The green boxes are the exported interfaces in which SPL programs are written, the blue boxes are the different syntaxes we manage internally and the red boxes are the execution targets.

## 8.1 Employed Haskell extensions

In this chapter, we are going to use some extensions to Haskell. The two major ones are generalized algebraic data types (*GADTs*) [PJVWW06] and *type families* [SPJCS08], both of which will be explained in this section.

### 8.1.1 GADTs

Ordinary algebraic data types in Haskell are defined via the `data` construct. Consider the following small language:

```
data Term
    = Constant Integer
    | Add Term Term
    | Pair Term Term
```

The data constructors represent integer constants, integer addition and a constructor for 2-tuples. Unfortunately we may construct terms that are nonsensical given this interpretation. For example,

```
Add (Pair (Constant 1) (Constant 2)) (Constant 3)
```

Since integer addition is not defined for pairs, we might like to prevent the construction of such terms. Adding a type parameter to `Term` gets us some of the way:

```
data Term a
    = Constant Integer
    | Add (Term Integer) (Term Integer)
    | Pair (Term x) (Term y)
```

In reality, the `x` and `y` type variables used in the `Pair` constructor would require a separate extension to enable *existential types* and an explicit quantifier. However, there is no way to specify, for example, that when `Constant` is given an `Integer` it returns a `Term Integer`. If `Constant` was a function, we might express that as follows:

```
Constant :: Integer -> Term Integer
```

In fact, this is the exact syntax that GADTs provide. We can now specify a typed syntax tree as follows:

```
data Term a where
    Constant :: Integer -> Term Integer
    Add      :: Term Integer -> Term Integer -> Term Integer
    Pair     :: Term x -> Term y -> Term (x, y)
```

Thus it's no longer possible to construct a term that tries to do integer addition on pairs, because `Pair` returns a term of pair type and the type of `Add` requires terms of integer type. We can use this type information when writing our interpreter:

```
interpret :: Term a -> a
interpret (Constant i) = i
interpret (Add x y) = interpret x + interpret y
interpret (Pair x y) = (interpret x, interpret y)
```

Note that we never refer to the type parameter `a` in the definition of `Term`. Just as data constructors can be thought of as functions, type constructors can be thought of as functions on the *type level*. A *kind* is the "type of a type". For types like `Integer` and `Bool`, the kind is `*`. For type constructors with one parameter, the kind is `* -> *`, because when given a type of kind `*` it produces another type of kind `*`. For a type constructor with two parameters, the kind is `* -> * -> *`, and so on and so forth. Our final definition of `Term` is as follows:

```
data Term :: * -> * where
    Constant :: Integer -> Term Integer
    Add      :: Term Integer -> Term Integer -> Term Integer
    Pair     :: Term x -> Term y -> Term (x, y)
```

### 8.1.2 Type families

Type families take the idea of type level functions one step further, by not being limited to the construction of types. Later on we are going to use *heterogeneous lists* constructed from iterated pairs and `()` as a terminator. For example, the type of a two-element list might be `(((), Int), Bool)`. We could construct the type level function that finds the type of the leftmost element (`Int` in this case):

```
type family LeftmostOf :: * -> *
type instance LeftmostOf ((), b) = b
type instance LeftmostOf (a, b)  = LeftmostOf a
```

Type families can be used in most places where an ordinary type can be used. Along with type families comes *equality constraints*, written `a ~ b`. For example, the following two constraints hold:

```
LeftmostOf (((), Int), Bool)  ~  Int
LeftmostOf ((), String)       ~  String
```

Like type class constraints, equality constraints can be written in any context. We will use this ability later on. The `Of` suffix will be used in type family names to make it easy to spot where they are used.

Type families can be associated with type classes. In that case, the type family declaration is placed inside the type class declaration and the instance declaration is placed inside the type class instances, with the `family` and `instance` qualifiers dropped for type families. They are slightly restricted for the purpose of better error messages, but the details are not important for understanding this chapter.

## 8.2 High level code

In order to do compilation at a later stage, we need to capture all of the computational steps required to sample from the distributions. One way of representing computation is to specify it in a programming language. Indeed, the internal representation of `Dist` and `Process` are syntax trees comprising a small language for stochastic computation. The main definitions are given in module `Language.SPL.Syntax`, while the operators are given in module `Language.SPL.Operator`.

```
data Dist :: * -> * where
    Normal  :: Dist Real
    Uniform :: Dist Real
    Lookup  :: Dist Time -> Process a -> Dist a
    Sample  :: Dist a -> (Dist a -> Dist b) -> Dist b
    Certain :: Constant a -> Dist a
    Unary   :: UnaryOperator a1 a2 ->
               Dist a1 -> Dist a2
    Binary  :: BinaryOperator a1 a2 a3 ->
               Dist a1 -> Dist a2 -> Dist a3
    Ternary :: TernaryOperator a1 a2 a3 a4 ->
               Dist a1 -> Dist a2 -> Dist a3 -> Dist a4
    TagD    :: Int -> Dist a

data Process :: * -> * where
    Closed  :: (Dist Time -> Dist a) -> Process a
    Prefix  :: (Dist Time -> Dist a -> Dist b -> Dist a) ->
               Dist a -> Process b -> Process a
    Zip     :: Process a -> Process b -> Process (a, b)
    Trace   :: Process a -> (Process a -> Process b) -> Process b
    TagP    :: Int -> Process a -> Process a
```

The `Normal`, `Uniform`, `Lookup`, `Sample`, `Closed`, `Prefix`, `Zip`, `Skip` and `Trace` constructors correspond exactly to their namesakes in lowercase from section 7.1.1.

`Certain` are for constants of type `Real` and `Bool`. `Unary`, `Binary` and `Ternary` are for built-in unary, binary and ternary operators respectively. These include the if-statement and the constructor and destructors for pairs.

The `Closed` and `Prefix` constructors accept arbitrary functions from distributions to distributions. This is called higher order abstract syntax, or HOAS, and is also used in Nikola and Accelerate. It allows us to keep a type preserving one-to-one mapping between the functions we provide and the representation we use. In turn, we can preserve these types as we transform the syntax tree and translate it to (typed) lower level code.

Having functions in the tree may look like a problem because we can't directly look under the arrow of a function, which would be necessary in order to translate it to a language outside Haskell. However, it's possible to *reify* such functions given some internal representation of variables in the syntax tree. Consider a syntax tree for untyped lambda calculus:

```
data Term = Lambda String Term | Apply Term Term | Variable String
```

Given a function `f :: Term -> Term`, applying `f` to any `Term` will obviously yield a syntax tree we can inspect. To reify the function, we can apply `f` to a *fresh* variable, and then wrap it in a lambda abstraction:

```
let body = f (Variable "x") :: Term in
Lambda "x" body :: Term
```

By doing this simple eta-expansion, we arrive at an inspectable representation of the function. In our syntax tree, `TagD` and `TagP` serves roughly the same purpose as `Variable` above. We will return to their exact use in section 8.4.

The `skip` function is implemented as a transformation on the syntax tree. Recall that it skips a given amount of time into a process:

```
skip :: Dist Time -> Process a -> Process a
skip t process = case process of
    Closed f -> Closed (\t' -> f (t + t'))
    Prefix f d0 p ->
        Prefix f (Lookup t (inclusivePrefix f d0 p)) (skip t p)
    Zip p1 p2 -> Zip (skip t p1) (skip t p2)
    Trace p f -> Trace p (skip t . f)
    TagP tag p -> TagP tag (skip t p)
```

For the `Closed` process that is directly a function from time to a distribution, we simply add the skip time to the input time. For prefix we skip by fast forwarding the initial value by the skip time, as well as skipping on the process prefixed over. The remaining cases simply recursively skip on their process arguments.

We have chosen to use `Double` as the discretization of `Real`, mainly for its availability on modern GP-GPUs:

```
type Real = Double
```

### 8.2.1 A running example

In order to give an intuition of the various representations and translation steps that we will look at in this chapter, we will now go through the translation of `lookup 5 brownian`, but first recall the definition of `brownian`:

```
brownian = iterative (\dt w -> w + normal * sqrt dt) 0
```

To make the representation readable, we're going to use Haskell's `where` clause to give names to subterms – to arrive at the actual representation, simply perform substitution. The following is the code generated in the high level language by the prelude functions:

```
brownian5 = Lookup (Certain (Double 5)) outer
    where
```

The `inclusivePrefix` (see section 7.1.2) used in the definition of `iterative` has introduced an outer `Prefix` and pairs for delaying the stream of values by one time step:

```
outer      = Prefix first undefined inner
inner      = Prefix f initial undefined
```

The initial value `0` is also duplicated by `inclusivePrefix`:

```
first _ _  = Unary First
initial    = Binary Pair zero zero
zero       = Certain (Double 0)
```

The iterated function `\dt w -> w + normal * sqrt dt` is straightforward, except that it does not use the value of the (`undefined`) process:

```
f dt w _   = Binary Pair (Unary Second w) (add w dt)
add w dt   = Binary Add  (Unary Second w) (product dt)
product dt = Binary Mult (Unary Sqrt dt)  Normal
```

We will return to this example later in this chapter and translate it further until we reach our target, the OpenCL code.

## 8.3 Low level code

In order to split up the compilation into smaller tasks, we introduce an intermediate language that is slightly closer to the target language. The idea is to make sampling via a pseudo random number generator (*PRNG*) and looping explicit at this level, removing the concept of processes and distributions. Additionally we would like to reify functions at this point, so that they can be inspected when doing the final translation to OpenCL code.

Before we can continue to the language itself, we must introduce one of the core concepts it uses, namely de Bruijn indexing.

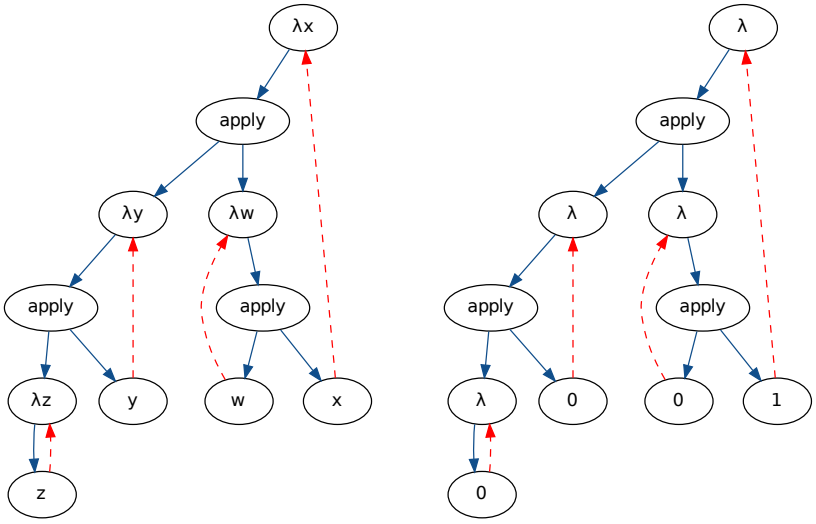
### 8.3.1 De Bruijn indexing

In the standard notation for lambda calculus, we may write down the following term:

$$\lambda x.\,(\lambda y.\,(\lambda z.\,z)\,y)\,(\lambda w.\,w\,x)$$

In this notation, the *binder* for a variable is the closest lambda in the syntax tree above the variable occurrence that mentions the variable name. This gives us a lot of choice in naming variables, and gives rise to *alpha-equivalence* between lambda terms. De Bruijn indexing [dB72] abolishes

variable names and instead uses natural numbers. These numbers indicate how many binders to skip when searching the syntax tree upwards to find the appropriate binder for a variable occurrence. This removes the choice in naming and makes checking for alpha equivalence a simple matter of checking for syntactic equivalence. Figure 8.2 compares the syntax trees and notations.




$\lambda x.\,(\lambda y.\,(\lambda z.\,z)\,y)\,(\lambda w.\,w\,x)$ $\qquad$ $\lambda\,(\lambda\,(\lambda\,0)\,0)\,(\lambda\,0\,1)$

Figure 8.2: Standard notation (on the left) vs. de Bruijn indexing (on the right). The dashed arrows connect variable occurrences to their binders.

A concrete benefit of the latter representation is that the environment can be a list of values, indexed by a natural number (the de Bruijn index), instead of the traditional finite map from names to values.

In particular, we can encode the *required environment* for an expression in the type system as a homogeneous list using iterated pairs . Consider the Haskell let-statement:

$$\texttt{let}\ x = e_1 :: a\ \texttt{in}\ e_2 :: b$$

We may introduce a type `Term env a` where `env` is the type of the required environment and `a` is the result type of the term. We can then introduce a let statement like the following:

```
Let :: Term env a -> Term (env, a) b -> Term env b
```

The variable name is nowhere in sight, since we assume de Bruijn indexing. Like Haskell's let statement, this would provide a binding to the first term during the evaluation of the second term. This is evident in the type, since the second term's environment type is extended with the result type of the first term. In other words, the environment is *typed*.

Our representation of environments uses nested pairs. The empty environment is represented by the unit type `()`. The `Term` type constructor is comparable to Haskell's function types in the sense that the types `a -> b` and `Term (env, a) b` both specify that a value of `b` can be produced given a value of type `a`. Some examples of this relationship are given in figure 8.3.

| Haskell type | Corresponding `Term` type |
|---:|---:|
| `c` | `Term () c` |
| `b -> c` | `Term ((), b) c` |
| `a -> b -> c` | `Term (((), a), b) c` |

Figure 8.3: Some examples of Haskell types and their corresponding `Term` types starting from an empty environment `()`.

As in [Cha09], we encode de Bruijn indexes as Peano numbers, with additional type information about the required environment and the variable type:

```
data Index :: * -> * -> * where
    Zero :: Index (env, a) a
    Succ :: Index env a -> Index (env, b) a
```

This `Index` type is used to index into an environment to find the appropriate value of the right type. At index `Zero`, the outermost pair of the environment must contain the appropriate value. When increasing the index by 1 with `Succ`, we're allowed to extend the environment with a value of any type of our choosing. We can use these indexes to perform lookups in the environment:

```
peek :: Index env a -> env -> a
peek Zero     (_,           a) = a
peek (Succ n) (environment, _) = peek n environment
```

The logic is that if the index is `Zero`, we're at the right position in the environment and we're guaranteed to find a value of the appropriate type there. If the index is `Succ n` then we peel off a layer of the environment and recursively `peek` into the smaller environment with a smaller index (ie. decreased by 1). We will use this function later on.

### 8.3.2 Low level syntax tree

We will now define the syntax tree of the low level code, which will be the result of the translation from high level code described in the next section. The definition is given in module `Language.SPL.Intermediate`.

#### The `Intermediate` data structure

The low level syntax tree is defined as the data type `Intermediate env a`. Note that there is an extra parameter `env` compared to the the high level syntax tree. This is because terms in the low level language may contain variables. The `env` is the type of environment required to provide values for these variables:

```
data Intermediate :: * -> * -> * where
```

The uniform and normal distributions, constants, unary, binary and ternary[1] operators are still built into this language:

```
  Uniform, Normal :: Intermediate env Double
  Constant        :: Constant a ->
                     Intermediate env a
  Unary           :: UnaryOperator a b ->
                     Intermediate env a ->
                     Intermediate env b
  Binary          :: BinaryOperator a b c ->
                     Intermediate env a ->
                     Intermediate env b ->
                     Intermediate env c
  If              :: Intermediate env Bool ->
                     Intermediate env a ->
                     Intermediate env a ->
                     Intermediate env a
```

The `env` specifies that any environment can be used here.
The language has strict let bindings (using de Bruijn indexing):

```
  Let             :: Intermediate env a ->
                     Intermediate (env, a) b ->
                     Intermediate env b
  Variable        :: Index env a ->
                     Intermediate env a
```

In addition, we have operators for splitting the current PRNG and storing one of them for later use via `Split` and `Use`:

[1] The if statement is the only ternary operator we have.

```
Split :: Intermediate (env, StdGen) a -> Intermediate env a
Use :: Index env StdGen -> Intermediate env a -> Intermediate env a
```

We will return to the exact use for these operators in section 8.4.

We will use the "`S.`" prefix to distinguish elements of the high level code from those of the low level code.

Finally, we have an accumulating loop. The `Prefix` name alludes to that these loops always stem from one or more `S.Prefix`es in the high level code:

```
Prefix :: Bool ->
          Intermediate env S.Time ->
          Accumulator env a ->
          Intermediate env a
```

The first argument keeps track of whether or not the current time is referred to anywhere in the process, which will be used later on. The second parameter is the lookup time, which is used to calculate the number of iterations to take in the loop. The third argument will be explained below.

### The `Accumulator` data structure

When a `S.Prefix` accumulates over another `S.Prefix` in the high level code, such as taking the `average` of a `brownian`, we don't want to incur a quadratic performance penalty by creating a nested loop. Instead we would like to run the two prefixes in lockstep, taking one iteration of the inner `S.Prefix` and then use the value to take one iteration of the outer `S.Prefix`, thus creating only a single loop. For this purpose we create a data structure to represent nested loops:

```
data Accumulator :: * -> * -> * where
```

The most complicated constructor is the one describing a `S.Prefix` from the high level code:

```
Accumulate ::
    Intermediate (((env, S.Time), a), b) a ->
    Bool ->
    Maybe (Intermediate env a) ->
    Maybe (Accumulator env b) ->
    Accumulator env a
```

The first argument is the body of the accumulator function from `S.Prefix`. It requires an environment containing the $\Delta t$, the accumulated value so far and the current value of the accumulated process. The second argument

records whether or not the $\Delta t$ argument is used. The third and fourth argument is the initial value and accumulated process respectively, or `Nothing` when they are not in use.

Processes that have been `S.Zipped` has a direct representation in this data structure:

```
Zip ::
    Accumulator env a ->
    Accumulator env b ->
    Accumulator env (a, b)
```

When we know the lookup time, any `Closed` process is simply a distribution, which gets translated to `Intermediate` as:

```
Expression ::
    Bool ->
    Intermediate (env, S.Time) a ->
    Accumulator env a
```

The boolean argument again keeps track of whether or not the current time is referred to inside the process.

Finally we have `Splitting` and `Using` which are analogous to `Split` and `Use` respectively:

```
Splitting ::
    Accumulator (env, StdGen) a ->
    Accumulator env a

Using ::
    Index env StdGen ->
    Accumulator env a ->
    Accumulator env a
```

We are now ready to translate the high level code into this representation.

## 8.4 Translation from high level to low level code

The translation code is provided in module `Language.SPL.Intermediate`.

We are going to use the same approach as Accelerate [Cha09] in order to convert HOAS to de Bruijn indexing. In particular, a lot of care is needed when constructing `TagD`, because the index used is a plain `Int` and thus guarantees nothing about the type of value it refers to. In contrast, the `Variable` uses a type safe index (temporarily stored in the `Layout`) so, if we do the conversion right, we will preserve the types from the HOAS representation.

In order to minimize surface area of this hazard, we introduce the following function which ensures that the tag created has the right `Int` index pointing to a corresponding `Index` of the right type in the `Layout`:

```
distTag ::
    Layout env env' ->
    (S.Dist a, Layout (env, a) (env', a))
distTag layout =
    (S.TagD (size layout), increase layout `Push` Zero)
```

This ensures that a tag created is always matched by a type safe index entry in the layout of the same type. As long as we use this layout in the right place, we are safe. A similar function is defined for `TagP`.

In order to get the type safe index out of the layout again, a `project` function is provided. Given that the layout is constructed correctly, it's safe to use, and will otherwise result in a runtime error.

### 8.4.1 Distributions

The conversion is done using a function that takes a layout and a distribution and returns the corresponding low level code:

```
convert' :: Layout env env -> S.Dist a -> Intermediate env a
convert' layout term = case term of
```

In the case of `Sample`, we use the strict let binding from the intermediate code. This ensures that we have performed the side effect of picking a random element from the distribution, and thus use the same value everywhere the variable is used:

```
    S.Sample e f ->
        Let (convert' layout e) (convert' layout' (f tag))
        where
            (tag, layout') = distTag layout
```

The tag introduced by this is handled by projecting the `Int` to an `Index env a` for use in the `Variable`:

```
    S.TagD tag ->
        Variable (project (size layout - tag - 1) layout)
```

Except for `S.Lookup`, the rest of the cases are either straightforward recursive cases or one to one mappings. In the case of `S.Lookup` we need to translate a `S.Process a`, which is done by a separate function that is covered in the next to sections:

```
    S.Lookup e p ->
        lookup layout e p
```

### 8.4.2 Simple lookups

Since the `lookup` function has a lookup time, it can get away with converting the otherwise infinite stochastic process into a simple sampling of a value in many cases:

```
lookup :: Layout env env -> S.Dist S.Time -> S.Process a -> Intermediate env a
lookup layout time process = case process of
```

For the `S.Closed` case, it is a simple matter of performing the lookup directly, by treating the process as a function from the lookup time to a distribution:

```
    S.Closed f ->
        convert' layout (f time)
```

The `S.Zip` case is a simple recursive case.

The `S.Trace` case is more involved. We have to ensure that everywhere the traced process is used, we get to look at the same time series. Assuming a PRNG with mutation, our approach to ensuring this is to:

- Create a new PRNG state from the current PRNG state.
- Store the new PRNG state in the environment.
- Inline the traced process at all usage sites.
- Use a fresh copy of the stored PRNG state whenever the execution reaches these.
- Use stack semantics to return to the previous PRNG state when the execution reaches the end of the code for the inlined process.

Only part of this approach is visible in the translation, since the PRNG state is a runtime value. The key thing to notice here is that `S.TagP` contains the process to be inlined and requires a PRNG state to be available in the environment:

```
    S.Trace p f ->
        Split   -- Will create the new PRNG state
            (lookup layout' time (f tag))
        where
            (tag, layout') = processTag layout p
    S.TagP tag p ->
        Use     -- Will use the stored PRNG state locally
            (project (size layout - tag - 1) layout)
            (lookup layout time p)
```

The last case is when we have a `S.Prefix`. In this case, we might need to introduce a loop to iterate or accumulate over a process. We introduce an `accumulator` function that builds a tree of nested applications of `S.Prefix`, in order to emit efficient loop code later on. The prefix case is as follows:

```
p@(S.Prefix _ _ p') ->
    Prefix
        (usesTimeInProcess p')
        (convert' layout time)
        (accumulator layout p)
```

Note that we perform a search in the tree here to discover whether or not the current time is referred to anywhere. This information is utilized later in the translation from low level code to OpenCL code.

### 8.4.3 Lookups on accumulating processes

The `accumulator` function looks like:

```
accumulator :: Layout env env -> S.Process a -> Accumulator env a
accumulator layout process = case process of
```

In this function, the case for `S.Prefix` is the most complicated. It requires that we can look into the body of the accumulating function to see which of the variables are used:

```
examine :: (Dist a -> Dist a1 -> Dist a2 -> Dist a3) -> (Bool, Bool, Bool)
examine f =
    let (x1, x2, x3) = (-4, -3, -2) in
    let body = f (TagD x1) (TagD x2) (TagD x3) in
    (varUsedInDist x1 body, varUsedInDist x2 body, varUsedInDist x3 body)
```

Note that the negative indexes and tags created here don't escape from the function. We can now implement the case for `S.Prefix`:

```
S.Prefix f e p ->
```

First we convert the initial value and the accumulated process:

```
    let accumulator' = accumulator layout p in
    let initialValue = convert' layout e in
```

Then we create tags for the three parameters for the accumulator function, and use these in the conversion of the function itself:

```
let (timeTag,       layout')   = distTag layout in
let (accumulateTag, layout'')  = distTag layout' in
let (valueTag,      layout''') = distTag layout'' in
let f' = convert' layout''' (f timeTag accumulateTag valueTag) in
```

Then we analyse which of the variables the function uses:

```
let (useDt, useAccumulator, useProcess) = S.examine f in
```

Finally we use a `Maybe` type to distinguish between used and unused variables when creating the `Accumulate` node:

```
Accumulate
    f'
    useDt
    (if useAccumulator then Just initialValue else Nothing)
    (if useProcess     then Just accumulator' else Nothing)
```

The `S.Zip`, `S.Trace` and `S.TagP` cases are recursive and analogous to the ones in `lookup`.

When we reach `S.Closed`, we note whether or not the current time is used in the process, create a tag for it to perform a `lookup`:

```
p@(S.Closed _) ->
    let (tag, layout') = distTag layout in
    Expression (usesTimeInProcess p) (lookup layout' tag p)
```

This concludes the translation from high level code to low level code. However, we might like some of the parameters of the distribution to be decided dynamically, rather than statically fixing them during the translation. This is the topic of the next section.

### 8.4.4 Top level functions of arbitrary arity

Since compilation takes time, it's beneficial to be able to reuse compiled code. In order to do this, we translate top level functions rather than just distributions, so that the compiled code can be run with different parameters. We will thus translate high level code functions with types like the following:

```
Dist a -> Dist b
Process a -> Dist b
Dist a -> Process b -> Dist c -> Dist d
```

Ie. any function with distribution and process parameters that yields a distribution in the end. Note that the parameters can only be instantiated to *values* – it's thus not possible to supply new code after the translation.

We introduce the following type class, whose purpose is to extend `convert'` to work on n-ary functions:

```
class Translate env a where
    translate' ::
        Layout env env ->
        a ->
        Intermediate (EnvironmentOf env a) (ResultOf a)
```

The `EnvironmentOf` and `ResultOf` are type families, respectively yielding the environment type and result type of low level code that is translated from high level code:

```
type family EnvironmentOf env a
type family ResultOf a
```

The base case is the plain distribution, which requires nothing of its environment:

```
instance Translate env (S.Dist a) where
    translate' = convert'
```

The environment for such a distribution has no extra requirements, and the result type is the the same as that of the sample space of the distribution:

```
type instance EnvironmentOf env (S.Dist a) = env
type instance ResultOf (S.Dist a) = a
```

Given that we can translate something, we can also translate it with one additional distribution parameter:

```
instance Translate (env, a) b => Translate env (S.Dist a -> b) where
    translate' layout f = translate' layout' (f tag)
        where
            (tag, layout') = distTag layout
```

Note that we add the parameter type to the environment type. The result type is that of the recursive case:

```
type instance EnvironmentOf env (S.Dist a -> b) = EnvironmentOf (env, a) b
type instance ResultOf (S.Dist a -> b) = ResultOf b
```

The case for process parameters is similar.

### 8.4.5 Low level code for the running example

Recall the example from section 8.2.1, namely `lookup 5 brownian`. The following is the translation to the low level language. To make it readable, we will pretty print the type safe de Bruijn `Index`es used in `Variable` as natural numbers.

The lookup allows us to fix the end time of the loop:

```
brownian5' = Prefix True (Constant (Double 5)) outer
    where
```

The outer prefix is a `map`, and thus doesn't use its initial value or $\Delta t$. The inner prefix uses $\Delta t$ and an initial value, but does not need a process:

```
outer   = Accumulate first False Nothing (Just inner)
inner   = Accumulate f True (Just initial) Nothing
```

The function that just takes the first element of the value from the inner process is now simply an expression with a free[2] variable:

```
first   = Unary First (Variable 0)
```

The initial value is basically identical to the high level code:

```
initial = Binary Pair zero zero
zero    = Constant (Double 0)
```

The iterated function is also quite similar, but now it's just the function body with free variables:

```
f       = Binary Pair (Unary Second (Variable 1)) add
add     = Binary Add  (Unary Second (Variable 1)) product
product = Binary Mult (Unary Sqrt   (Variable 2)) Normal
```

Note how the syntax tree is now a first order data structure with no functions blocking the view from what's going on inside.

## 8.5 OpenCL device architecture

OpenCL is a standard for massively parallel computation [Khr08]. It is supported by massively parallel hardware (GP-GPUs) from both AMD and NVIDIA, and is similar to the NVIDIA-specific CUDA platform.

Figure 8.4 shows a simplified model of the OpenCL device architecture. Every *device* has a large *global memory* that can be accessed by all threads.

[2]Not completely free, since it's recorded in the environment type.

The threads are partitioned into *work groups*, each group running on a single *compute unit*. When a thread executes, it executes on one of the *processing elements* within this compute unit. The threads in a work group share the vastly smaller *local memory*.

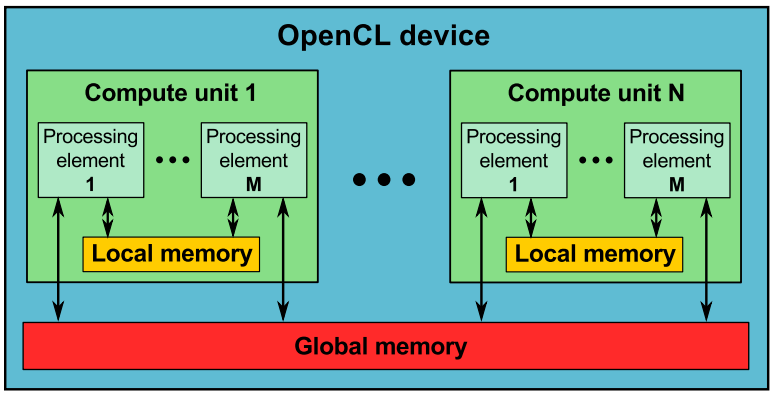



Figure 8.4: Simplified OpenCL device architecture.

The *host*, in our case the CPU running the Haskell program, is responsible for allocating global and local memory and starting the threads. The threads themselves can't allocate any additional memory and can't start additional threads.

The computation model is a so called single instruction, multiple data model, better known as *SIMD*. In particular, threads executing simultaneously on the same compute unit execute the same instructions. This is possible because all threads execute the same program. If the execution of some threads diverge, such as when they are in different branches of an if-statement or loops for more iterations, then the other threads are simply suspended while the first ones finish diverging.

The entry point in the program that all threads execute is called a *kernel*. It's written in a C-like language with certain restrictions and extensions; in particular, both function pointers and recursion are disallowed [Khr08].

A small program may look like this:

```
kernel void f(global float* a, local float* b, float c) {
    size_t global_work_size             = get_global_size(0);
    size_t global_thread_id             = get_global_id(0);
    size_t local_work_size              = get_local_size(0);
    size_t thread_id_within_work_group  = get_local_id(0);
    // ...
}
```

The `kernel` keyword specifies that this function is an entry point. Such a function must return `void`. The first argument is a pointer to some float-

ing point numbers in global memory, as indicated by the `global` keyword. Similarly, the `local` keyword specifies that the second argument is a pointer to something in local memory. The last argument is a single value that is neither in local nor global memory. There is no restriction on the number or order of arguments, but any pointer arguments to a kernel must be declared to belong to a specific *memory region*, such as the global or local memory.

There may be multiple functions in the source code, and any number of those may be marked as kernels. It's also possible to include code via the `#include` directive, if the appropriate include directories are specified when compiling the source code.

Certain features are only available via extensions, such as IEEE 754 double precision floating point numbers, which can be enabled by the `#pragma OPENCL EXTENSION cl_khr_fp64 : enable` directive.

When executing a kernel from the host, it is necessary to specify the total number of threads that will be executed, called the *global work size*, as well as the number of threads in each thread group, called the *local work size*. For each pointer argument to the kernel it is also necessary to specify how much memory to allocate. Global memory can also be initialized from the host before executing the kernel, as well as copied back into the host memory after the kernel has finished executing.

These numbers can be queried from within the thread, as is evident from the code example. The threads also have a global and a local ID, queried with `get_global_id(0)` and `get_local_id(0)` respectively[3].

Tasks like kernel execution and data transfer is performed *asynchronously*. For each device, any number of *event queues* can be created, and tasks are enqueued into these queues. When enqueueing tasks you can specify a list of events to wait for before the task executes. Tasks fire a completion event when they are done executing, and these events can be used to order data transfers and execution of kernels.

There are several other memory regions, such as *registers* and *private memory*, which are local to the individual process elements and used at the discretion of the compiler when no memory region is specified. It should also be noted that depending on the implementation, access to global memory may be cached, but in general it can be assumed that access to local memory is much faster than to global memory.

On most OpenCL targets, accessing shared memory in specific patterns is required in order to get optimal performance. Our approach is to bypass such difficulties simply by not using shared memory at all for the bulk of the computation.

To use OpenCL from Haskell we use the HOpenCL library developed by Martin Dybdal (unpublished), which is a one-to-one mapping of the OpenCL C API to Haskell functions, providing a friendlier and less error

[3]The 0 specifies the first dimension, which is the only for the purpose of this thesis.

prone interface (eg. using lists instead of the (size, pointer) pairs required in the C API). However, the library uses finalizers for resource management, which is problematic because the resources on the device may run out before the finalizers are executed. By choosing to use this library, we unfortunately inherit this weakness.

## 8.6 Translation from low level code to OpenCL code

This section describes the translation from low level code to OpenCL code. The code for this translation step is given in module `Language.SPL.OpenCL.Compiler`.

### 8.6.1 Quasi quotation for C-like languages

In order to generate OpenCL code, we have extended the C *quasi quoter* [Mai07] used in Nikola to support OpenCL's variant of C. Like in Nikola, this allows us to write C code in special quotation marks, while splicing in code and values generated elsewhere in the Haskell program. Syntax errors for the C code is then reported at Haskell compile time[4].

### 8.6.2 Preserving (some) typing with phantom types

Since the C syntax tree we generate is untyped, we can't be sure that the generated code will type check, even if the syntax tree we generate it from is fully typed. However, to get some help from the compiler, we use phantom types for variable names and C expressions:

```
newtype Name a = Name String
newtype Expression a = Expression Exp
```

Note that the type variables are not used on the right hand side of the definitions. At runtime, these are just `String` and `Exp` respectively, but until then we can use the type variables to capture type information.

Since C (thankfully) uses names and not de Bruijn indexing for variables, we need to generate suitable names and connect them to our de Bruijn indexes. Our approach is to store the generated names in an environment indexable by the type safe de Bruijn indexes. For this purpose, we convert the iterated pair representation of the environment by wrapping each element in the `Name` type using the following data type family:

```
data family Named :: * -> *
data instance Named ()       = NamedEmpty
data instance Named (env, b) = NamedBind (Named env) (Name b)
```

[4]C type errors are not caught until OpenCL compile time, however.

Creating a modified version of `peek` that can perform lookups in the name environment is straightforward:

```
peek :: Index env a -> Named env -> Name a
peek Zero     (NamedBind _ a)           = a
peek (Succ n) (NamedBind environment _) = peek n environment
```

A key advantage of these phantom types is that we can define the following type preserving functions for working with variables:

```
assign :: Name a -> Expression a -> M ()
bind   :: String -> Expression a -> M (Name a)
use    :: Name a -> M (Expression a)
```

These are for generating C code for destructively updating, creating and initializing a variable, and getting the C expression that uses a variable respectively. Note that the `String` in `bind` is for supplying a friendly prefix for the variable name; a unique suffix is generated automatically.

The `M` monad is used for collecting generated statements, keeping track of used variables and pair types, and generating fresh suffixes for variable names. It's the `Writer` monad transformed with `StateT`.

### 8.6.3 The simple cases of `Intermediate`

The translation of constants and unary and binary operators is a straightforward one-to-one translation. Recall that the `If` statement is *strict in all its arguments*; we use the `select` OpenCL intrinsic to implement this.

The `Let` and `Variable` constructs insert and look up names in the environment respectively, and map to the corresponding C constructs. We generate unique names and declare all variables in the top of the generated code block, so there is little to think of in terms of scoping.

### 8.6.4 The primitive distributions `Uniform` and `Normal`

To generate (pseudo) random samples from the `Uniform` and `Normal` distributions we have employed the MWC64X [Tho11] pseudo random number generator for OpenCL. Like most PRNGs, it samples from an uniform distribution of integers, in this case the $2^{32}$ first natural numbers. We convert this to a standard uniform distribution between 0 and 1 by division. We then use the Box-Muller method [GM58] to convert these to standard normal distributions.

### 8.6.5 The `Split` and `Use` constructs

The compiler keeps a stack of C variable names referring to PRNG states internally. The topmost variable on the stack is used whenever a call to the PRNG is generated. The `Split` construct uses the state from the current topmost variable to generate a new PRNG state by randomization of the internal state, and stores it in a new variable. The `Use` construct performs a deep copy (by assignment) of this new variable and pushes it onto the stack for the translation of its body.

### 8.6.6 The `Accumulator` loops

As discussed in section 8.3.2, we want to run nested `prefix`es in lockstep rather than generating nested loops whenever possible. Most of that work was already done when generating the `Accumulator` trees, since everything in such a tree can run in lockstep. However, we would like to go even further and move as much code out of the loop as possible, ideally removing the loop altogether. This is crucial optimization because every use of `lift`, `lift2`, etc. introduces a `Prefix` in the high level code, and thus creates an `Accumulate` node.

Every `Accumulator` node is in exactly one of the following categories. Note that we never reorder nodes within these categories, but only move the categories around as whole pieces:

**The nodes that must be inside a loop.** If the node is an `Accumulate` node and it mentions the accumulator variable, then we (likely) cannot move it out of the loop, because the accumulator variable may change every iteration. If the node sits between two nodes that cannot be moved out of the loop, then it also (likely) cannot be moved out of the loop, because it's a function of input that (likely) changes each iteration, and a fresh result is (likely) needed every iteration.

**The nodes that can be moved to after the loop.** If this node is not in the first category, and no nodes above it is in the first category, then it can be moved to after the loop because only the last result it generates will ever be used, and nothing it depends on can be moved below it.

**The nodes that can be moved to before the loop.** If this node is not in the first category, but a node above it is in the first category, then it can be moved to before the loop given that it does not depend on anything that varies over time. Otherwise it must stay inside the loop, because a fresh result is (likely) needed every iteration. The only two things that varies over time in this context (and which are not in the first category) are the current time, `Uniform` and `Normal`.

We move the nodes according to these rules. Consequently, if the first category is empty, then the loop is empty and can be eliminated entirely.

The example in figure 8.5 has nodes in all three categories (colored black, blue and red respectively).

Whenever we generate a loop for a stochastic process, we must choose a $\Delta t$. A preferred time step has already been supplied to the compile function at this point, but the end time for the loop might not be divisible by this time step. We find the largest number not greater than the time step that divides the end time and use that in the loop.

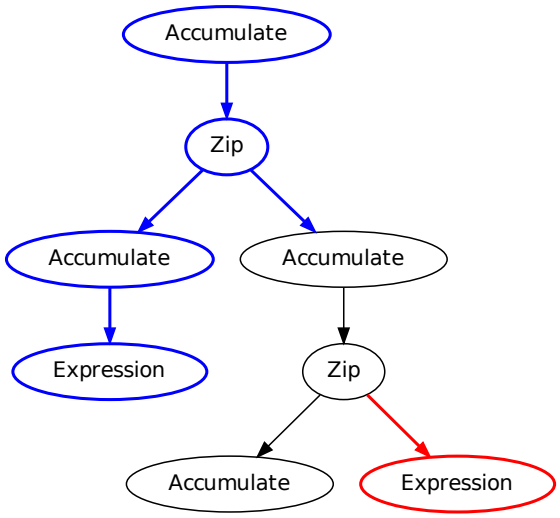



Figure 8.5: Example of an `Accumulator` tree. If node $a$ depends on node $b$, there's an arrow from $a$ to $b$. Assume that the black `Accumulate` nodes require the accumulator variable; then they must stay in the loop, and so must the black `Zip`. Assume that the blue nodes do not require the accumulator variable; then they can be moved to after the loop. Assume that the red nodes requires neither the accumulator variable nor the current time, and are not stochastic; then they can be moved to before the loop.

### 8.6.7 Wrapping it up

To the top of every generated program, we add a directive enabling the double precision floating point type:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

We include the code for the PRNG and for taking the mean and standard deviation:

```
#include "random.cl"
#include "sum.cl"
```

For every pair type encountered, we generate a `struct` with two fields `first` and `second` with types corresponding to the first and the second element of the pair. The name of the struct is `pair_n1_n2_t` where `n1` and `n2` are the names of the types of the first and the second element respectively.

The kernel function itself takes a seed for the PRNG, shared memory buffers for emitting the final result, and then as many arguments as the compiled function had. Inside the function we initialize the time step and the PRNG, and declare all variables used. Then comes the code we generated for the stochastic computation, and as the last thing a call to `emit_result` with the result of the computation.

### 8.6.8 OpenCL code for the running example

As an example, recall the example from section 8.2.1, namely `lookup 5 brownian`. We generate the following OpenCL code[5]:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#include "random.cl"
#include "sum.cl"
struct pair_double_double_t {
    double first;
    double second;
};
kernel void simulate(ulong seed, local double* local_means, local
                     double* local_standard_deviations, global
                     double* global_means, global
                     double* global_standard_deviations)
{
    double time_step = 0.1;
    struct generator_t generator = initialize(seed);
    struct pair_double_double_t accumulator_5;
    double accumulator_6;
    double delta_0;
    double end_1;
    double normal_7;
    double step_3;
```

[5]This is the verbatim output of `compile 0.1 (lookup 5 brownian)` with the `printDebug` flag on in `Language.SPL.OpenCL.Runner`.

```
    double steps_2;
    double time_4;

    end_1 = 5.0;
    time_4 = end_1;
    steps_2 = ceil(end_1 / time_step);
    delta_0 = steps_2 == 0 ? time_step : end_1 / steps_2;
    accumulator_5 = (struct pair_double_double_t) {0.0, 0.0};
    time_4 = -delta_0;
    for (step_3 = 0; step_3 <= steps_2; step_3 += 1) {
        time_4 += delta_0;
        normal_7 = normal(&generator);
        accumulator_5 = (struct pair_double_double_t) {accumulator_5.second,
                                                       accumulator_5.second +
                                                       sqrt(delta_0) *
                                                       normal_7};
    }
    accumulator_6 = accumulator_5.first;
    emit_result(accumulator_6, local_means, local_standard_deviations,
                global_means, global_standard_deviations);
}
```

The code contains the calculation of the $\Delta t$ to best fit the preferred time step. Then there is the actual loop, which generates a new sample from the normal distribution each iteration and updates the accumulator variable for the Brownian motion accordingly. The pair type that was introduced by `inclusivePrefix` has a type declaration and uses the literal syntax for struct values, as well as field accessors. One of the optimizations from section 8.6.6 is (barely) evident in this simple example; the code for what was originally `map first` has been moved out of the loop.

## 8.7 Execution on the GP-GPU(s)

Since we want to run functions with any number of parameters, we need to take another look at converting function types. The code for this section is in module `Language.SPL.OpenCL.Runner`. We introduce a type family for this purpose:

```
type family FunctionOf :: * -> *
```

When we have instantiated all parameters and are left with a distribution, all that's left is to execute the simulation and get the result back:

```
type instance FunctionOf (Dist a) = IO SimulationResult
```

Parameters can only be instantiated to values, which we encode in the type system by stripping away `Dist` and `Process` recursively:

```
type instance FunctionOf    (Dist a -> b) = a -> FunctionOf b
type instance FunctionOf (Process a -> b) = a -> FunctionOf b
```

We also do this translation at the value level. This involves building up a function corresponding to the type that converts each argument to the HOpenCL's `KernelArg` type, and builds the IO monad that executes the simulation in the base case. This is handled by the `Function` and `Arguments` type classes.

### 8.7.1 Execution of the kernels

Figure 8.6 shows the tasks that make up the execution of the Monte Carlo simulation. We perform the same steps on all OpenCL devices and then aggregate the results on the host in the end. First we run the simulation kernel, which runs the generated OpenCL code and computes the mean and standard deviation of each work group. Then we run the collector kernel which aggregates the work group mean and standard deviation to a device global mean and standard deviation. Finally we copy this result from the device to the host.

We use events in order to make sure that the results have been produced before they are consumed. In OpenCL, every enqueued task has a completion event, and every task can wait on any number of events before starting, so long as they are from the same event queue. The host can wait on events from all command queues, and does so before aggregating the results.

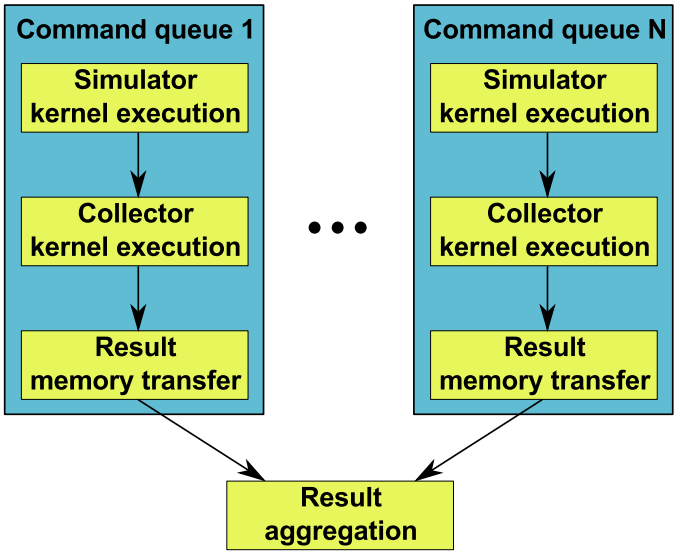



Figure 8.6: This figure shows the event scheduling. For every OpenCL device there is an OpenCL command queue (blue), in which kernel execution and memory transfer tasks are scheduled. The tasks (yellow) are ordered via events (arrows from source to listener).

### 8.7.2 Result aggregation

In the end, the results of each individual thread must be combined into a mean and standard deviation. This is done in parallel on the GP-GPU; first the thread of the simulator kernel cooperate to generate a mean and a standard deviation for each work group, and then another kernel is executed whose threads cooperate to compute a mean and a standard deviation by combining those of the work groups on the device. When multiple devices are used, the host does a similar combination of the results from the devices. The algorithm is sketched in figure 8.7.

The final result is a `SimulationResult` containing the mean and standard deviation. This data structure and related functions can be found in module `Language.SPL.SimulationResult`. The `Show` instance for this type produces the string `m ± s` where `m` is the mean and `s` is the standard deviation, as seen in section 1.2.

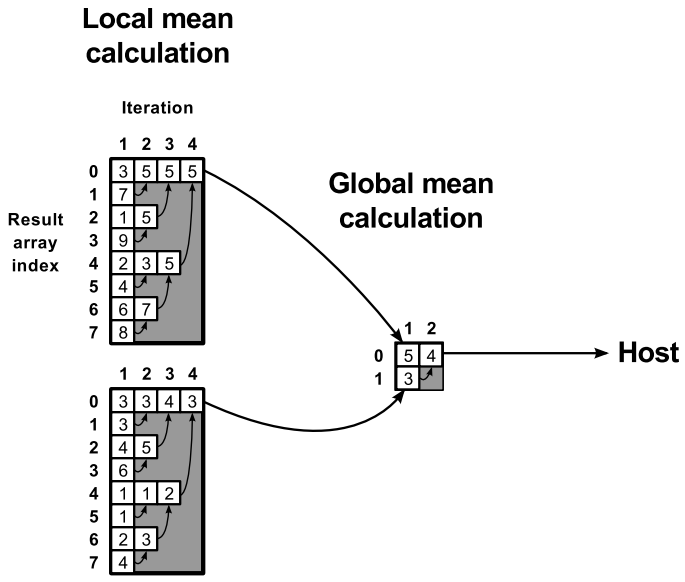



Figure 8.7: This figure shows the calculation of the mean for 2 work groups with 8 simulation results each. The reduction in local memory (on the left) and global memory (on the right) use the same merging algorithm: every iteration of the algorithm, every two adjacent values are combined into a single value (their mean). An array is used to hold the result, and over time less and less elements are in use (those that aren't are colored grey). The algorithm stops when there is only a single element left, at index 0. The result is then sent to the next aggregator; first to the device global aggregator and then to the final result aggregator on the host.

# Chapter 9

# Correctness

Using computer programs in the financial industry to price derivatives requires great confidence that the programs do in fact calculate what the programmer or domain expert had in mind. We have tried to accommodate this in the design of SPL. A SPL program is simply a model of a random variable written in a way that mimics the conceptual understanding of mathematical stochastic distributions and processes. The programmer should not be too concerned about writing these stochastic calculations efficiently as this is the job of the compiler. Furthermore, the programmer will never have to write the error prone code to carry out the simulation to find an expected value of the specified distribution, as this is the very evaluation semantics of SPL. We hope that these factors will add enough transparency between developed SPL programs and the underlying conceptual understanding such that the domain expert may be easily convinced, that the programs are in fact correct.

But program correctness is a little help if the language implementation is not, and this is the topic we are going to cover in this section. We would have liked to formally prove that our OpenCL implementation is a correct implementation of SPL. That is, to show that the C-code generated form the SPL syntax do in fact obey the SPL semantics when evaluated according to the OpenCL specification. This seems to be a serious challenge, which we have found our self unable to overcome and we have instead chosen to take an experimental approach to justify the correctness of our OpenCL SPL compiler.

## 9.1 Test strategy

We first of all want to test the correctness of the OpenCL back end for *SPL* as this is also a prerequisite for the correctness of the *CC* pricer using *SPL*. It is however also a concern, that the *CC* pricer do in fact produce the right *SPL* code. But we feel quite confident that this is the case, as it is almost

identical to the implementation given in the paper [JES00]. This is despite the fact that we recently discovered an error in this pricer, when pricing contracts having choices based on future values. This is so because we see the error as an result of a bad understanding of our own *SPL* semantics, rather than a result of producing the wrong pricing code.

We do on the other hand consider the OpenCL back end to be far more error prone, which is why we only do systematic testing of the *SPL* language.

We are using two Haskell test frameworks to carry out the tests. The first is QuickCheck [CH00], which is used to test that a function or parametric property is correct, given a reference implementation. The second one is HUnit, which is a Haskell clone of the JUnit framework, seen for Java. The HUnit tests are all simple, just testing that a calculated result matches the expected result.

We had actually hoped to integrate this test with Haskell cabal, such that the tests could be run using `cabal test`. But as the used GP-GPU memory is first freed by the time the Haskell finalizers run, we cannot include this many tests in one execution. This turned out to be a very annoying problem, as it forced us to run the tests in small chunks. These chunks may all be started individually from the Haskell module `Language.SPL.Test.Tests`.

The comparison of simulated results needs to tolerate some error margin, whenever the simulated distribution models uncertainties. We will, in this case, use a error margin of 1% of the largest two compared values.

The next section describes our systematic testing of all the language constructs, as well as some of the prelude functions. These tests are highly synthetic, trying to provoke errors to occur in the border cases of the various language constructs. The subsequent section, do in contrary, describe several more realistic pricing test where the known good results are drawn from either closed formulas or other financial papers. Section 9.4 describes a test that confirms the problem described in 7.2.1, that we cannot price contracts having a choice containing subcontracts looking into the future.

Our OpenCL SPL implementation does not have any known error at the time of writing and all the tests are running successfully, except for the test in section 9.4 showing our known error.

## 9.2 Structured language tests

The language unit tests are organized in seven groups holding more than a hundred small test cases in total. The seven groups are briefly covered below without mentioning of the trivial cases. The actual unit tests can be found in module `Language.SPL.Test.UnitTests` included in appendix D.

**Dist constructs**

This groups consist of eight tests, testing all the basic `Dist` constructs

except for the unary and binary operators. We first test that normal and uniform yields an expected value of 0 and 0.5 respectively. `Lookup` is tested utilizing that `\t -> lookup t time` is the identity on `Dist Double`. For `sample` it is tested that `uniform ‘sample‘ \d -> d - d` yields an expected value of exactly zero. We do also test that just subtracting two `uniform` from each other without sampling do not give an expected value of zero.

**Unary operators**
This test group tests each of the unary operators with one or two input values each. The input values are chosen such that one operator could not be potentially mistaken as another. The ternary `if_` is also tested here.

**Binary operators** As above this groups tests all the binary operators.

**Process constructs**
In this test group we test the `Always`, `Closed`, `Trace` and `Zip`. `Prefix` and `skib` are covered directly here but has gotten their own test groups because they are implemented using several optimizing special cases.

The OpenCL interpreter does not handle processes at top level which is why the tested processes have been prepended with a lookup at a specific time. The SPL `Process` AST does not contain operators on processes directly, but all the operators on distributions can be lifted to work pointwise on processes using `Prefix` and `Zip` and the pair type. This is why we do not test the operators exhaustively for processes but only provide a few cases for unary and binary operators and for the ternary if operator.

**Accumulator optimizations**
`Prefix` is a powerful and versatile construct. In its strongest form it can be used to implement aggregates like `integral` in a time step varying environment. It is also used to implement time step indifferent aggregates like `maximum` and the even simpler, but most important function `map`, that ignores both the delta time and the accumulator value.

The conversion from *SPL* to *Intermediate* detects which of the formal parameters in the accumulator function that are used in the function body. This information is then used in the OpenCL compiler to de-nest loops and to move loop invariant code out of the loops and to only calculate the current loop time when needed. This is why it is important to test different variations of using/ignoring the accumulator function parameters for various processes using nested `Prefix`'s.

We have not systematically exhausted all of these combinations as there are quite a few. This would be a good point to gain additional

confidence in the OpenCL implementation by writing a more complete set of test cases for `Prefix`.

**skip**

`skip` is not part of the *SPL* AST but it is not a prelude function either as it is implemented as a rewriting on the non-exposed *SPL* AST.

This test group tests skip in conjunction with various process constructions.

**Properties**

During the design and implementation of *SPL* the set of build-in constructs and prelude functions has changed a whole lot. In its current form *SPL* have gotten the general construct `Prefix` built-in, which is the backbone for implementing derived skeleton functions `map` and `iterative` but also the higher level functions `integrate` and `brownian`. The current `Prefix` handle the initial accumulator values as if it was a value from the past. While this favour the implementation of `map` it makes the implementation of `iterative` somewhat more involved and it is definitely something to keep in mind when implementing functions like `integrate`.

But regardless of which functions that is implemented using which and of the way we chose to handle delta times in the various functions, we have had a few properties in mind though out the process that wanted to provide. These properties are tested here:

**Prefix at time zero:**
`lookup 0 (prefix f d0 p)` ≡ `f dt d0 (lookup 0 p)`

**PrefixInclusive at time zero:**
`lookup 0 (prefixInclusive f d0 p)` ≡ `d0`

**Brownian starts at zero:**
`lookup 0 brownian` ≡ `constant 0`

**Integrate p is zero at time 0:**
`lookup 0 (sum p)` ≡ `0`

**Integrate is correct on constant processes:**
`lookup t (sum (always k))` ≡ `constant k * t`

## 9.3 Pricing tests

While the test described in the last section was very focused on testing the individual constructs of *SPL* and the prelude exploiting implementations details when it seemed beneficial, the tests presented in this section takes an overall proof of concept approach to show that *SPL* can in fact be used for what it's designed for. This is of course to simulate the expected price for

a brood range of financial contracts. This is probably the test the financial experts would be more concerned about at it shows that we can find good estimated prices compared to known good results.

This test does not only serve to boost the confidence in the OpenCL back end's correctness, it also acts as a showcase of how to model well known contracts in *SPL*. We have tested four contract types which are covered in the next four sections.

### 9.3.1 Zero coupon discount bond

A zero coupon discount bond that at time $t$ pays out $k$ amount in DKK, can be expressed using the Composing Contract combinators like this:

```
zcb t k = get (truncate t (scale (constant amount) (one DKK)))
```

When priced using a constant continuous rate model with rate $r$ we should find the exact value $ke^{-rt}$. We have tested this property using Haskell's QuickCheck with $t \in [0; 20]$, $k \in \mathbb{R}^+$ and $r \in [0; 2]$.

The code can be found in module `Language.CC.Test.PricingTest`.

### 9.3.2 Underlying sanity check

To model the European call options presented in the next subsection we needed to model what is often referred to at a standard underlying based on a given initial price, a volatility and the risk free interest rate. Specifically this is the underlying that the closed form Black-Scholes formula subsumes. We asked Mogens Steffensen from the Mathematics department of Copenhagen University to provide us with such an underlying and he encouraged us to do this little test presented here.

The underlying is modelled as

$$Se^{(r-\frac{1}{2}v^2)t+v\mathcal{W}(t)}$$

where $t$ is the time, $S$ is the initial price, $r$ is the rate, $v$ is the volatility and $\mathcal{W}(t)$ is a brownian motion. A property that any sane underlying should full fill is:

> The current value of receiving an underlying at any future time $t$ is $S$

or formulated differently, on average, it should be worth the same to receive an underlying today or in the future.

The contract is expressed in *CC* while the underlying is expressed in *SPL*

```
underlying s r v =
    always s * exp (always (r - 0.5 * v ^ 2) * time
    + always v * brownian)

contract t s r v =
    get (truncate t (scale (underlying s r v)))
```

As earlier we use QuickCheck to test that the price of `contract t s r v` is in fact $s$.

The code can be found in module `Language.CC.Test.PricingTest`.

### 9.3.3 European call options

We have already seen how to model an European call options in *CC*. In this test we take such a contract price it that test the price against the closed Black-Scholes formula. Black-Scholes formula as well as the *CC* European call options contract are parametrized over the initial price, the strike price, the maturity time, the rate and the volatility. This is also the parameters in our QuickCheck property.

The code can be found in module `Language.CC.Test.PricingTest`.

### 9.3.4 Asian call options

The Asian call options is the first contract type for which we do not know a closed formula to calculate reference prices, and we have therefore searched the literature to find some reference prices. We are using the prices given in table 1 in [And98].

The payout of an Asian call options is calculated as the average of the underlying subtracted the strike price in a given period. The *CC* combinators cannot express such aggregation over processes in its current form, which is why our Asian call options is modelled purely in *SPL*. The time for using QuickCheck as also past, as we now only have reference prices for 9 configurations.

The code can be found in module `Language.SPL.Test.AsianTest`.

### 9.3.5 Lookback options

The lookback options are similar to the Asian options, except they use another aggregating function than the average; in this case the maximum. We test both fixed strike and floating strike lookback options, using the prices given in table 3 and 4 in [And98].

The code can be found in module `Language.SPL.Test.LookbackTest`.

### 9.3.6 Basket options

We price a basket option with two correlated underlyings and a variable rate model. Our SPL code is a port from the corresponding R program from [Ano10] and is compared to the results of that simulation. *CC* supports both multiple underlyings and a variable rate model, but we wanted to test SPL in isolation here.

The code can be found in module `Language.SPL.Test.BasketTest`.

## 9.4 Choice based on future value

We discussed in section 7.2.1 our problems involved in pricing contract giving the holder a choice that should be taken based on expected future values. The American options is highly based on this kind of choices, which is why we do not support the *CC* `anytime` combinator. But it is still possible to express *CC* contract which is not rejected by our pricer, but instead given a incorrect, unnatural high price. This test experimentally verifies the existence of this error in our *CC* pricer.

Consider the game that always pays the player \$2 with a 25% probability.

Then consider the observable where a 6 faced die is rolled at every point in time. The observable value is three whenever a six is rolled and zero otherwise. This is modelled in *SPL* as the process

```
die :: Observable Double
die = always $ choice (1/6) 3 0
```

The expected value of this observable is $\frac{1}{6} \cdot 3 = \frac{1}{2}$ at every point in time. Now consider the contract that gives you the choice of receiving \$1 or to get the value of `die` paid in dollars

```
cashOrPlay :: Contract
cashOrPlay = one USD `or` scale die (one USD)
```

The holder of this contract will not have a hard time to make up her mind, as the present value of an observables, per definition, is certain. The holder would therefore simple choose the \$3 when a six is rolled and take the 1\$ other wise. The value of the contract is therefore $\frac{5}{6} \cdot 1 + \frac{1}{6} * 3 = \frac{5+3}{6} = \frac{4}{3}$.

Now consider the slightly modified contract where the holder have the choose between 1\$ now or the contract that let the holder get the value of the `die` observable in the future.

```
cashOrPlayLater :: Contract
cashOrPlayLater =
    one USD `or` get (truncate 1 (scale die (one USD)))
```

Any rational holder would now choose the 1 dollar now as the expected future value of the other option only is \$0.5. The value of the modified contract is therefore \$1.

These are the two experiments carried out in this test. We test that the expected value of `payOrPlay` is $\frac{4}{3}$ and that the expected value of `payOrPlayLater` is 1 when prices in USD using the *SPL* pricer. The contract was prices using a zero discounting model. The expected price have been simulated on both of the interpreters and the OpenCL back. The first test ran successfully but the second test resulted in simulations results of around $1.33 \pm 0.75$ for all three simulators, which of course should have been $1 \pm 0$.

The result is not surprising at this point, as it confirms the problems discussed in section 7.2.1.

## 9.5 Summary

We have tested a range of base cases and combinations of slightly more complex cases from *SPL*, to capture isolated problems and to ensure that fixed bugs did not get reintroduced. We have tested pricers for European-style options, both vanilla and exotic (with path dependence and multiple sources of uncertainty), in part to show that *SPL* can be used to price a wide range of financial contracts, and in part to have complex examples with the potential to highlight bugs that only occur with a non-trivial combination of the language constructs. Indeed, the failing test case in section 9.4 is derived from a more complex example of pricing American options. The European call options were tested with QuickCheck with a wide range of parameters up against the known-good Black-Scholes formula, while the other pricers were only tested against the results we could find in the literature. Except for the one case mentioned, all tests pass (tested on fermi01, see figure 10.1).

# Chapter 10

# Benchmarks

While the previous section layed out the basis for the correctness of the OpenCL SPL back end, this section will focus upon the efficiency of the back end. This chapter presents four results from running four different benchmarks, all of them solely measuring the *execution time* from running SPL programs on the OpenCL back end. The justification for this is that we can compile a *function*, which can then be re-run with different parameters, thus being able to get arbitrarily many different results per compilation[1].

Our main result is that the simulation scales with the number of cores on the GP-GPU. We don't see any obstacles that would prevent us from using multiple devices, and the implementation has support for it, but we have not been able to verify this, and thus leave it to future work.

The second result is that we can efficiently overcome the simulation limit that prevents us from running more than about a million simulations per invocation. This is simply done by invoking the OpenCL kernel queue several times from Haskell and then combine the results.

Using the memory on the GP-GPU may severely slow things down when done improperly. The third result shows that computing the average result and standard deviation, which uses the local and global memory, only makes up about a fifth of the total scheduling/collection overhead.

Finally, we test the asymptotic performance of loop de-nesting and skipping, whose implementations are not entirely straightforward.

## 10.1 Hardware and software configurations

Figure 10.1 shows the hardware we have used to run the benchmarks on. We have used fermi01 for the majority of the benchmarks, but as this became unavailable near the end of the project, we have run some benchmarks on mem-beast1. Unless otherwise noted, fermi01 has been used.

[1]The compilation takes less than a second in all cases we have tried, except in the case of generating exponential code for the nested simulation.

| Hostname: | **fermi01** | **mem-beast1** |
|---|---|---|
| Devices: | 4×Tesla C2050 | 2×GeForce GTX 460 |
| OpenCL: | 1.0 CUDA | 1.0 CUDA |
| Driver version: | 275.21 | 260.19.26 |
| Compute units: | 14 | 7 |
| Total number of cores: | 448 | 336 |
| Work group size: | 1024 (maximum) | 1024 (maximum) |
| Global memory: | 2687 MiB | 1023 MiB |
| Local memory: | 48 KiB | 48 KiB |

Figure 10.1: Hardware and software configurations.

## 10.2 Scalability

It is common when testing whether a program scales, to investigate how the number of used cores influence the execution time for a fixed amount of work. In this scenario full scalability or utilization is achieved when doubling the amount of cores halves the execution time.

We do however not know how to tell OpenCL to only use a certain amount of the cores on the device but we can exploit the fact that threads in one work group all are scheduled on the same multiprocessor. The Tesla C2050 card used in our benchmarks have 14 multiprocessors on board which means that running a kernel with a work group count of 14 should take no longer than having only 1, if full utilization is meet. Running 15 work groups on the other hand would take twice as long as the last work group would have to wait until the other 14 groups had been executed in parallel. Fortunately, this is exactly the picture we see on in figure 10.2.

The experiment is carried out using an Asian Call option parametrized as indicated in the table below.

| Option type | Initial price | Rate | Volatility | Strike price |
|---|---|---|---|---|
| Asian call | 100 | 0.05 | 0.2 | 100 |

We have not chosen the Asian option for any particular reason. The Asian option runs only one loop in each simulation thread. In this case we are having an iteration for each day in fire years which is 1825 iterations as may be seen in the table below.

| Work group size | Group count | Multiplier | Time step | Lookup time |
|---|---|---|---|---|
| 512 | $\{2 \dots 42\}$ | 1 | 1/365 | 5 |

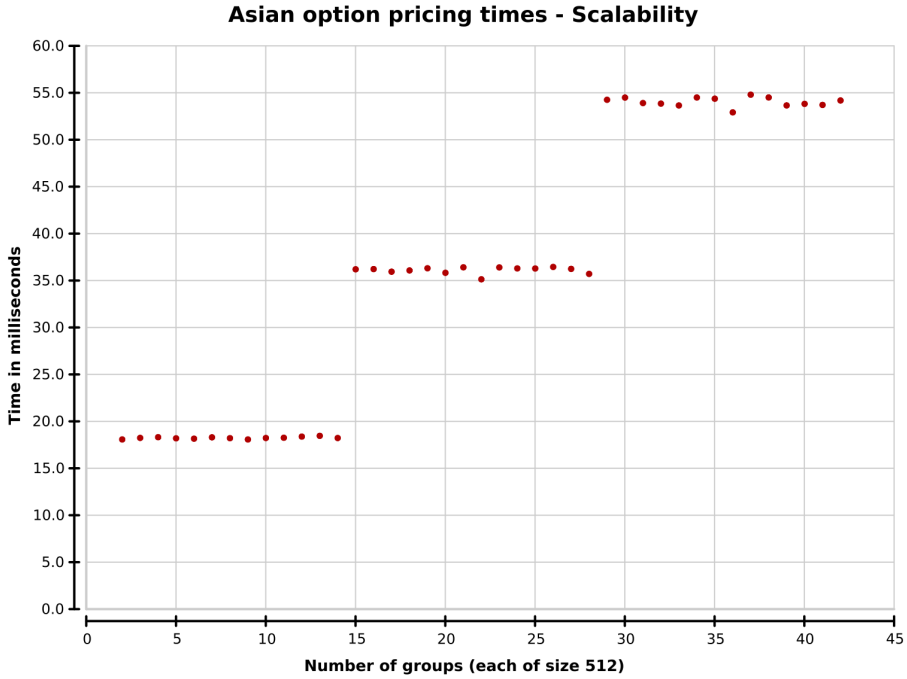



Figure 10.2: This graph shows the pricing times for an asian call option valuated using 1024 to 21504 simulations. The number of simulations are indexed by the work group count to show the correlation with the 14 multiprocessors on the Tesla C2050 device. The graph shows that we may add simulations from additional work groups without increasing the executions time until the point where the work groups count reach the next multiple of 14. Each pricing experiment have been performed 101 times. The minimum, maximum and quartiles of this data are shown in figure A.1.

It is not surprising to achieve full scalability as Monte Carlo simulations is *embarrassingly parallel* in nature.

## 10.3 How far can we go

The way we calculate the expected value and standard deviation on the GP-GPU limits us to have no more than about a million threads per kernel invocation on the Tesla C2050 card. Even though it might not be too complicated to overcome this limitation, we have first tried the simpler solution of simply invoking the simulation kernel queue setup several times. We do not need to compile and load the SPL program more than once so we expected the cost of doing several invocations to be relative low compared to

the overall execution time.

In the test setup we have yet again chosen the Asian option used in the previous section.

| Option type | Initial price | Rate | Volatility | Strike price |
|---|---|---|---|---|
| Asian call | 100 | 0.05 | 0.2 | 100 |

The Tesla C2050 card do not allow us to have more than 2048 work groups when using a work group size of 512 which is why this the maximum number of simulations we are going to run per invocation. So to go beyond that limit we try the experiment with different multipliers which indicate how many OpenCL invocations we perform. This configuration and more are specified in the table below.

| Work group size | Group count | Multiplier | Time step | Lookup time |
|---|---|---|---|---|
| 512 | $\{64, 128 \ldots 2048\}$ | $\{1, 2, 4, 8\}$ | 1/365 | 0.5 |

Figure 10.3 shows that the overhead from doing multiple invocations drowns in the actual computation time. This result confirms our intuition that the limit on the number simulations per invocation is not a practical problem.

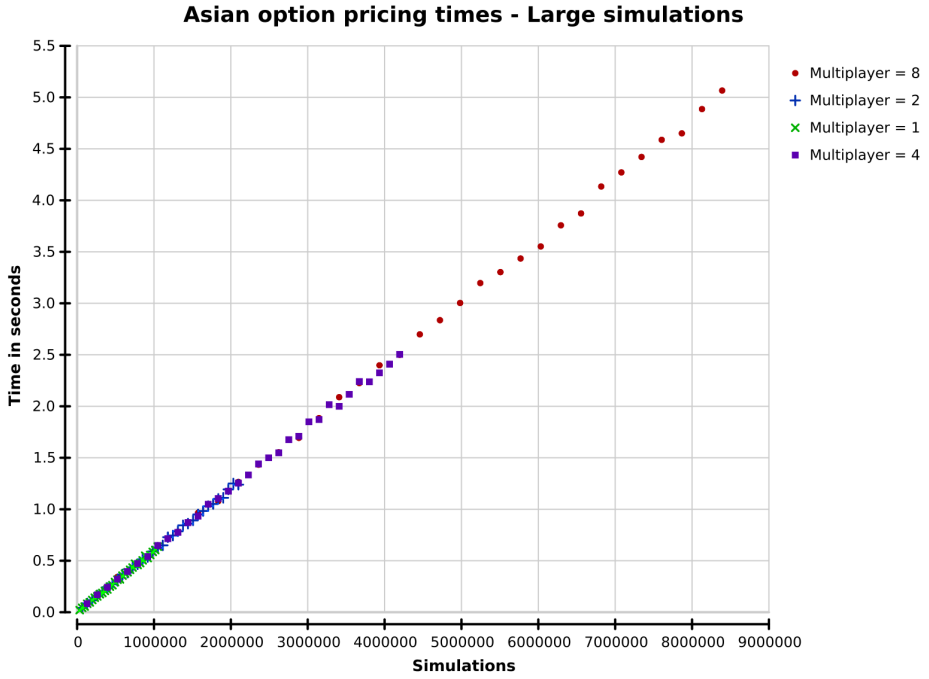



Figure 10.3: Using a fixed group size of 512 while running up to 2048 groups together with a multiplier of 8 yields more the 8 millions simulations. The overhead from invoking the OpenCL kernel several times is not visible when running $2^{20}$ simulations in each invocation. The SPL program in use is an Asian call option expiring after 6 month with a time step of one day. Each pricing experiment have been performed 101 times. The minimum, maximum and quartiles of this data are shown in figure A.2, A.3, A.4 and A.5.

## 10.4 Scheduling and result gathering overhead

There is some overhead inherent in scheduling the threads on the GP-GPU and gathering the results. Figure 10.4 uses a process that does almost no work, which shows how the overhead changes as a function of the number of simulations. It also shows that the overhead involved in the cooperative calculation of the mean and standard deviation is about 22% of the total overhead. Note that the overhead is negligible when the work load is small.

Beware that the data for this one graph was produced on mem-beast1.

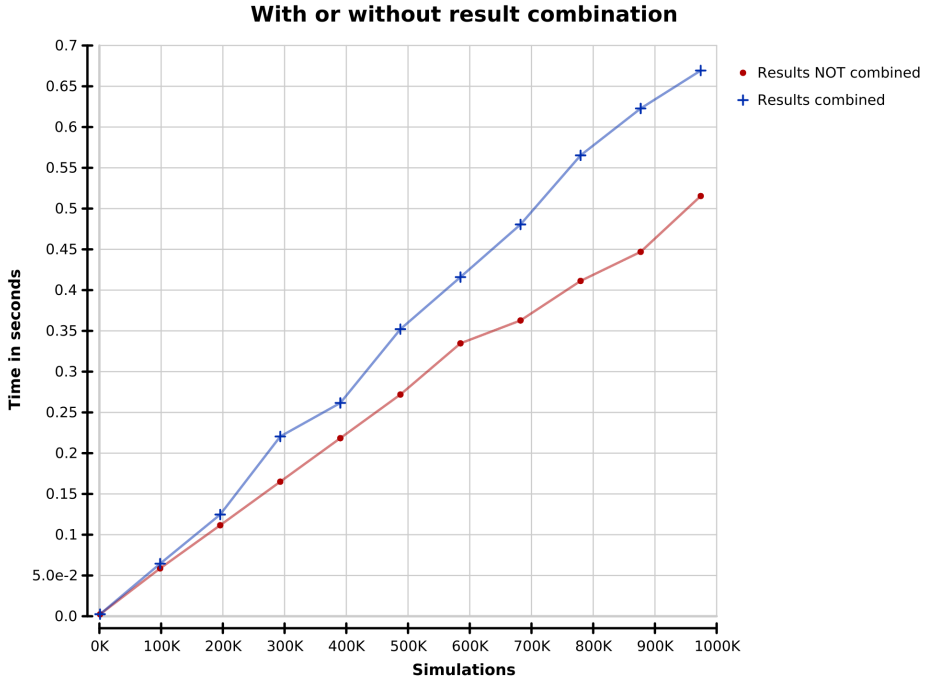


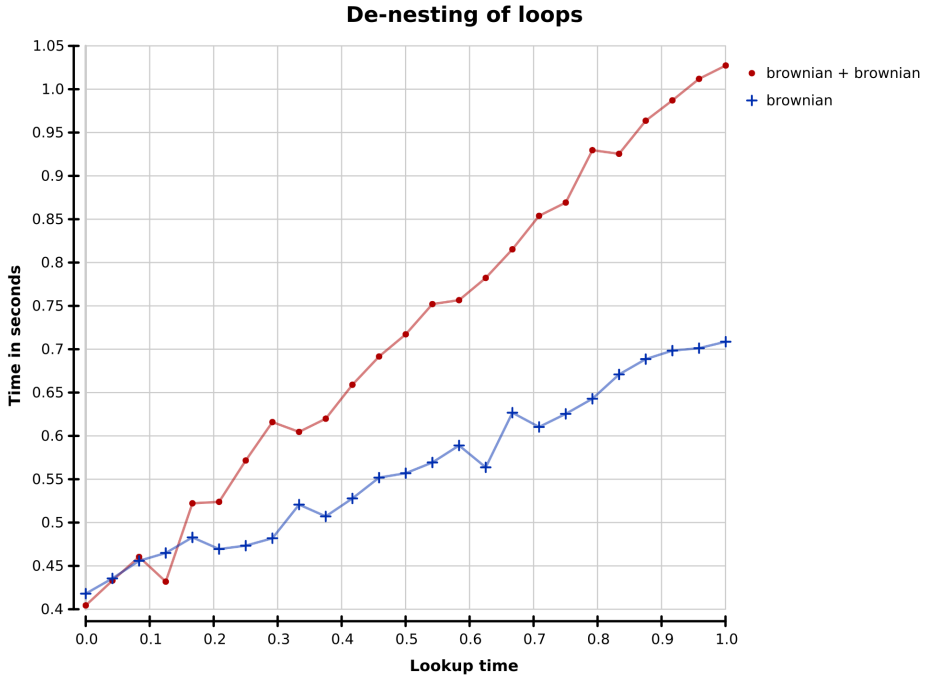

Figure 10.4: Scheduling overhead, tested using the process `always 42`. Each pricing experiment have been performed 11 times. The minimum, maximum and quartiles of this data are shown in figure A.6

## 10.5 Performance of selected SPL constucts

### 10.5.1 De-nesting of loops

When a `prefix` is nested inside a `prefix`, naive code generation will create a loop inside a loop. As discussed in section 8.6.6, we combine arbitrarily nested loops into a single loop, except that `closed` is never crossed.

To see that this is indeed the case and that we get the expected performance, we test with the following two processes: the first one is `brownian`, and the second one is `brownian + brownian`. The `+` uses `lift2` which uses `prefix` and `zip` internally. Disregarding constant-time overhead, if the de-nesting works in this case, we should should expect the time consumption of the `brownian + brownian` to be approximately twice of that of `brownian`. If it doesn't work in this case, we should expect the difference to be quadratic, since we will have a loop inside a loop. As can be seen in figure 10.5, the de-nesting works in this case.




| Work group size | Group count | Multiplier | Time step | Lookup time |
|---|---|---|---|---|
| 512 | 2048 | 1 | 1/365 | $\{0, 1/24 \ldots 1\}$ |

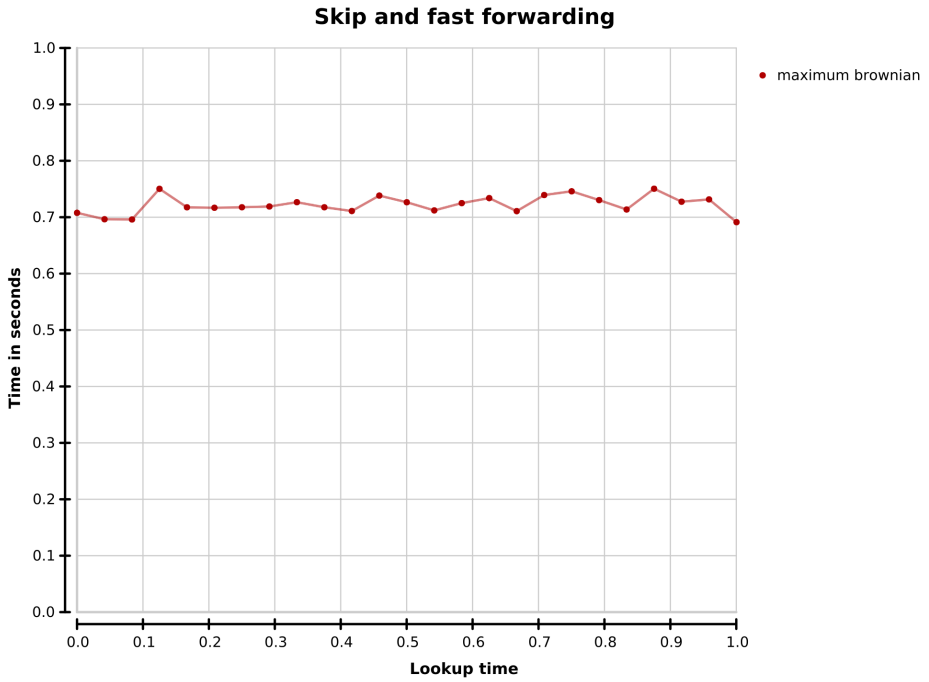
| Plot Label | SPL Process |
|---|---|
| brownian | `brownian` |
| brownian + brownian | `brownian + brownian` |

Figure 10.5: Simulation time for de-nesting of loops. Each pricing experiment have been performed 11 times. The minimum, maximum and quartiles of this data are shown in figure A.7.

### 10.5.2 Skip

As discussed in section 8.2, skip works by fast-forwarding the process by a specified amount of time. This means that for processes involving `prefix`, skip will run a loop to get to the point in time that was skipped to. A function that uses this is `maximum_`, which takes the maximum of a time interval by first skipping to the early point in the time interval. Since the time used in `max_` is dwarfed by the PRNG, we should then see that regardless of the starting time, the time consumption remains the same, since the fast forwarding interval and `max_`'ed interval always adds up to the full interval,

and this is indeed what we observe in figure 10.6.




| Work group size | Group count | Multiplier | Time step | Lookup time |
|---|---|---|---|---|
| 512 | 2048 | 1 | 1/365 | $\{0, 1/24 \ldots 1\}$ |

| Plot Label | SPL Process |
|---|---|
| maximum brownian | `maximum_ brownian` |

Figure 10.6: Simulation time for `skipping`. Each pricing experiment have been performed 101 times. The minimum, maximum and quartiles of this data are shown in figure A.8.

# Chapter 11

# Future work

American and Bermudan options can't be priced with SPL, as we have seen in section 7.2. This is due to the semantics of SPL, which assumes that future events can be known in the present. The holder of a financial contract obviously has no such crystal ball, but instead needs to base her choices on what happens *on average*. One interpretation of this is, that we need nested simulations, but more efficient strategies may be available in some cases. In order to complete the *CC* pricer, this must be investigated.

It would be interesting to see how *CC* could be extended to support Asian and Lookback-like options. One possibility is to add aggregates over time intervals directly, but then there's the question of how to specify aggregating functions like the average, maximum, etc. A finite list of functions would probably be too limiting.

As reported in chapter 9, we have tested pricers written in CC and SPL against results from the literature. However, while the tests for vanilla options are systematic, the tests for exotic options have only been tested with a small range of parameters for which we could find comparable results. In order to be confident of the correctness of both the pricers and SPL itself, it should be systematically tested. QuantLib [ABB+11] is a mature library with another approach to pricing financial contracts, and could serve as a reference pricer, against which to perform automated testing.

The benchmarks in chapter 10 measure scalability and the comparative performance of different SPL programs. It would be interesting to see how well the Monte Carlo simulating SPL implementation performs compared to state of the art pricing libraries like QuantLib.

As noted in chapter 8.5, our implementation does not deterministically clean up the resources it allocates on the GP-GPU, due to the use of finalizers. This is a problem because the GP-GPU is likely to run out of resources before the host program performs a garbage collection that triggers finalization. This needs to be solved, both at the level of the library we use for OpenCL, and at the level of the interface for compiling and running SPL

programs.

Our implementation does not quite match the semantics of SPL, since it doesn't use a uniform discretization of time. The numerical stability of the implementation has also not been studied, and it could have consequences for the accuracy of the result. To improve the confidence in the implementation, it would be useful to test the implementation against the semantics. Since the semantics is given as a Haskell program, it should be possible to test against this directly by comparing the expected value of the executable semantics with that of the GP-GPU implementation. One way of doing this is to generate SPL programs using QuickCheck's `Arbitrary` instance, although some thought will have to be put into generating interesting SPL program candidates. Simulation would probably also needed for the semantics in order to make this approach feasible.

# Chapter 12

# Conclusion

Our main contribution has been a domain specific language, called *SPL*, for stochastic processes in a financial pricing setting. We have analyzed the requirements for such a language, designed the language as an embedded language in Haskell with very little notational overhead, given it a type system in terms of Haskell's type system, given it semantics in terms of the probability monad and given it a Monte Carlo simulating parallel implementation that scales on the GP-GPU.

We have presented benchmark data showing this linear scaling with the number of available cores, and indicating that our loop de-nesting results in the expected performance for loops. Additionally, we have tested the *SPL* constructs in isolation and in complex combinations corresponding to actual use cases, giving an indication of correctness of the implementation.

We have applied several advanced techniques in order to achieve a smooth embedding of *SPL* into Haskell. We used type classes to provide operator overloading, allowing arithmetic on distributions and processes, using standard arithmetic notation. Regular functions and bindings can be used with *SPL* and we preserve this type safe representation using HOAS, and perform a type preserving translation to first order intermediate code with de Bruijn indexing. We have used type families in the translation of a function producing a *SPL* distribution, allowing us to compile a parametric OpenCL kernel once, and then run it multiple times by calling the compiled function.

*SPL* is not bound to the GP-GPU architecture or Monte Carlo simulation as the numerical method. We have given *SPL* executable semantics in terms of a Haskell interpreter, using an interface to the probability monad. One can easily obtain an *SPL* implementation from this interpreter, by using any of the other discussed probabilistic functional programming libraries.

We have used *SPL* to price both path dependent contracts and contracts whose price depends on multiple sources of uncertainty. Monte Carlo simulation is the preferred way to price both of these types of contracts, which motivates the need for such a back end.

We have shown an example implementation of the financial model component of *CC*. Although realistic financial models are not our domain, we conjecture that a realistic financial model component for *CC* could equally well be implemented using *SPL*.

With the prices for path dependent contracts, we have gone beyond the contracts expressible in *CC*. It would thus be possible to extend *CC* to support these and still use *SPL* for the pricer.

At the same time, *CC* can express contracts that can't be priced in *SPL*, namely American/Bermudan-style options, ie. any contract where the holder has to make a choice of *when* to exercise. We have identified the problematic part of the semantics of *SPL*, but have left solution for future work.

# Bibliography


[ABB+11] Ferdinando Ametrano, Luigi Ballabio, Marco Bianchetti, Nicolas Di Césaré, Dirk Eddelbuettel, Neil Firth, Nicola Jean, Chris Kenyon, Roland Lichters, Marco Marchioro, Klaus Spanderen, and Joseph Wang. QuantLib. `http://quantlib.org/index.shtml`, 2011.

[And98] Jesper Andreasen. The pricing of discretely sampled asian and lookback options: a change of numeraire approach. 1998.

[Ano10] Anonymous. Basket option pricing: Step by step. `http://stotastic.com/wordpress/2010/05/basket-option-pricing/`, 2010.

[BBG97] Phelim Boyle, Mark Broadie, and Paul Glasserman. Monte Carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21(8-9):1267–1321, June 1997.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *In Proceedings Supercomputing '90*, pages 666–675, 1990.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[Cha09] Manuel M. T. Chakravarty. Converting a hoas term gadt into a de bruijn term gadt. `http://www.cse.unsw.edu.au/~chak/haskell/term-conv/`, 2009.

[CKL+11] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *DAMP*, pages 3–14, 2011.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[EK06a] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[EK06b] Martin Erwig and Steve Kollmansberger. Modeling genome evolution with a dsel for probabilistic programming. In *PADL*, pages 134–149, 2006.

[Gil09] Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, 09/2009 2009.

[Gir82] Michèle Giry. As cited by Erwig and Kollmansberger, 2006. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin / Heidelberg, 1982. 10.1007/BFb0092872.

[GM58] George and Mervin E. Muller. A note on the generation of random normal deviates. *Ann. Math. Stat.*, 29(2):610–611, 1958.

[Hul09] J. Hull. *Options, futures and other derivatives.* Prentice Hall finance series. Pearson/Prentice Hall, 2009.

[ISO99] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[JE03] S. L. Peyton Jones and J-M. Eber. How to write a financial contract, 2003.

[JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. ACM.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* http://haskell.org/, September 2002.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[Kid07] Eric Kidd. Build your own probability monads, 2007.

[Lar11] Ken Friis Larsen. Implementing probability monads efficiently. Unpublished. `http://diku.dk/~kflarsen/t/ProbMonad-unpublished.pdf`, 2011.

[LS01] Francis A Longstaff and Eduardo S Schwartz. Valuing american options by simulation: A simple least-squares approach. *Review of Financial Studies*, 14(1):113–47, 2001.

[Mai07] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82, New York, NY, USA, 2007. ACM.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

[NVI07] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[PJVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

[SME99] Simon, Simon Marlow, and Conal Elliott. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *Implementation of Functional Languages*, pages 37–58, 1999.

[SPJCS08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43:51–62, September 2008.

[Tho11] David B. Thomas. The MWC64X random number generator. `http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-mwc64x.html`, 2011.

# Appendix A

# Benchmark data

Section 10 presents several graph showing the execution time for various pricing experiments. These experiments have all been performed several times where the graphs show the median of these results. The tables presented below provides the minimum, maximum and the three quartiles of the value of these experiments.

| Number of groups (each of size 512) | |
|---|---|
| 1024 | $1.79{\cdot}10^{-2}/1.81{\cdot}10^{-2}/\mathbf{1.82{\cdot}10^{-2}}/1.82{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 1536 | $1.77{\cdot}10^{-2}/1.81{\cdot}10^{-2}/\mathbf{1.82{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.86{\cdot}10^{-2}$ |
| 2048 | $1.79{\cdot}10^{-2}/1.81{\cdot}10^{-2}/\mathbf{1.82{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 2560 | $1.76{\cdot}10^{-2}/1.81{\cdot}10^{-2}/\mathbf{1.82{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 3072 | $1.79{\cdot}10^{-2}/1.81{\cdot}10^{-2}/\mathbf{1.82{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 3584 | $1.8{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.84{\cdot}10^{-2}/2.71{\cdot}10^{-2}$ |
| 4096 | $1.79{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.86{\cdot}10^{-2}$ |
| 4608 | $1.8{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.84{\cdot}10^{-2}/1.86{\cdot}10^{-2}$ |
| 5120 | $1.79{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.89{\cdot}10^{-2}$ |
| 5632 | $1.8{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 6144 | $1.79{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.84{\cdot}10^{-2}/1.87{\cdot}10^{-2}$ |
| 6656 | $1.78{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.84{\cdot}10^{-2}/3.18{\cdot}10^{-2}$ |
| 7168 | $1.79{\cdot}10^{-2}/1.82{\cdot}10^{-2}/\mathbf{1.83{\cdot}10^{-2}}/1.83{\cdot}10^{-2}/1.85{\cdot}10^{-2}$ |
| 7680 | $3.53{\cdot}10^{-2}/3.59{\cdot}10^{-2}/\mathbf{3.61{\cdot}10^{-2}}/3.62{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 8192 | $3.55{\cdot}10^{-2}/3.59{\cdot}10^{-2}/\mathbf{3.61{\cdot}10^{-2}}/3.62{\cdot}10^{-2}/3.66{\cdot}10^{-2}$ |
| 8704 | $3.55{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.64{\cdot}10^{-2}/3.68{\cdot}10^{-2}$ |
| 9216 | $3.55{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.61{\cdot}10^{-2}}/3.63{\cdot}10^{-2}/5.01{\cdot}10^{-2}$ |
| 9728 | $3.54{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.63{\cdot}10^{-2}/3.68{\cdot}10^{-2}$ |
| 10240 | $3.57{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.61{\cdot}10^{-2}}/3.63{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 10752 | $3.54{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.64{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 11264 | $3.51{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.63{\cdot}10^{-2}/3.68{\cdot}10^{-2}$ |
| 11776 | $3.54{\cdot}10^{-2}/3.61{\cdot}10^{-2}/\mathbf{3.63{\cdot}10^{-2}}/3.65{\cdot}10^{-2}/3.68{\cdot}10^{-2}$ |
| 12288 | $3.55{\cdot}10^{-2}/3.62{\cdot}10^{-2}/\mathbf{3.63{\cdot}10^{-2}}/3.65{\cdot}10^{-2}/3.71{\cdot}10^{-2}$ |
| 12800 | $3.56{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.63{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 13312 | $3.56{\cdot}10^{-2}/3.6{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.64{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 13824 | $3.55{\cdot}10^{-2}/3.61{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.64{\cdot}10^{-2}/3.67{\cdot}10^{-2}$ |
| 14336 | $3.54{\cdot}10^{-2}/3.61{\cdot}10^{-2}/\mathbf{3.62{\cdot}10^{-2}}/3.64{\cdot}10^{-2}/3.68{\cdot}10^{-2}$ |
| 14848 | $5.3{\cdot}10^{-2}/5.37{\cdot}10^{-2}/\mathbf{5.4{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.62{\cdot}10^{-2}$ |
| 15360 | $5.3{\cdot}10^{-2}/5.38{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.47{\cdot}10^{-2}$ |
| 15872 | $5.28{\cdot}10^{-2}/5.39{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.52{\cdot}10^{-2}$ |
| 16384 | $5.29{\cdot}10^{-2}/5.39{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.5{\cdot}10^{-2}$ |
| 16896 | $5.29{\cdot}10^{-2}/5.38{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.48{\cdot}10^{-2}$ |
| 17408 | $5.33{\cdot}10^{-2}/5.4{\cdot}10^{-2}/\mathbf{5.42{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.52{\cdot}10^{-2}$ |
| 17920 | $5.33{\cdot}10^{-2}/5.38{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.43{\cdot}10^{-2}/5.49{\cdot}10^{-2}$ |
| 18432 | $5.29{\cdot}10^{-2}/5.4{\cdot}10^{-2}/\mathbf{5.42{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.51{\cdot}10^{-2}$ |
| 18944 | $5.3{\cdot}10^{-2}/5.38{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.49{\cdot}10^{-2}$ |
| 19456 | $5.31{\cdot}10^{-2}/5.39{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.5{\cdot}10^{-2}$ |
| 19968 | $5.34{\cdot}10^{-2}/5.4{\cdot}10^{-2}/\mathbf{5.43{\cdot}10^{-2}}/5.45{\cdot}10^{-2}/5.52{\cdot}10^{-2}$ |
| 20480 | $5.31{\cdot}10^{-2}/5.39{\cdot}10^{-2}/\mathbf{5.41{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.52{\cdot}10^{-2}$ |
| 20992 | $5.33{\cdot}10^{-2}/5.4{\cdot}10^{-2}/\mathbf{5.42{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.48{\cdot}10^{-2}$ |
| 21504 | $5.32{\cdot}10^{-2}/5.39{\cdot}10^{-2}/\mathbf{5.42{\cdot}10^{-2}}/5.44{\cdot}10^{-2}/5.5{\cdot}10^{-2}$ |

Figure A.1: Minimum, maximum and quartiles for the scalability graph in figure 10.2

| Simulations | Multiplayer = 1 |
|---|---|
| 32768 | $2.04{\cdot}10^{-2}/2.07{\cdot}10^{-2}/\mathbf{2.15{\cdot}10^{-2}}/2.2{\cdot}10^{-2}/2.25{\cdot}10^{-2}$ |
| 65536 | $4.08{\cdot}10^{-2}/4.2{\cdot}10^{-2}/\mathbf{4.33{\cdot}10^{-2}}/4.36{\cdot}10^{-2}/4.5{\cdot}10^{-2}$ |
| 98304 | $5.72{\cdot}10^{-2}/5.85{\cdot}10^{-2}/\mathbf{5.93{\cdot}10^{-2}}/6.01{\cdot}10^{-2}/6.19{\cdot}10^{-2}$ |
| 131072 | $7.79{\cdot}10^{-2}/7.92{\cdot}10^{-2}/\mathbf{8.1{\cdot}10^{-2}}/8.3{\cdot}10^{-2}/8.5{\cdot}10^{-2}$ |
| 163840 | $9.69{\cdot}10^{-2}/9.71{\cdot}10^{-2}/\mathbf{9.86{\cdot}10^{-2}}/0.101/0.105$ |
| 196608 | 0.116/0.116/**0.121**/0.123/0.126 |
| 229376 | 0.131/0.135/**0.137**/0.141/0.143 |
| 262144 | 0.154/0.158/**0.16**/0.163/0.166 |
| 294912 | 0.17/0.176/**0.177**/0.182/0.185 |
| 327680 | 0.189/0.196/**0.199**/0.201/0.205 |
| 360448 | 0.207/0.209/**0.22**/0.222/0.225 |
| 393216 | 0.223/0.227/**0.233**/0.238/0.238 |
| 425984 | 0.251/0.255/**0.257**/0.26/0.264 |
| 458752 | 0.265/0.273/**0.277**/0.283/0.285 |
| 491520 | 0.284/0.29/**0.295**/0.303/0.314 |
| 524288 | 0.301/0.315/**0.322**/0.325/0.331 |
| 557056 | 0.315/0.326/**0.332**/0.338/0.355 |
| 589824 | 0.339/0.349/**0.356**/0.358/0.363 |
| 622592 | 0.361/0.366/**0.37**/0.377/0.383 |
| 655360 | 0.375/0.377/**0.391**/0.4/0.413 |
| 688128 | 0.399/0.406/**0.414**/0.429/0.434 |
| 720896 | 0.416/0.425/**0.438**/0.442/0.454 |
| 753664 | 0.429/0.451/**0.461**/0.469/0.479 |
| 786432 | 0.454/0.461/**0.466**/0.484/0.488 |
| 819200 | 0.479/0.485/**0.493**/0.506/0.506 |
| 851968 | 0.48/0.496/**0.505**/0.525/0.529 |
| 884736 | 0.515/0.523/**0.53**/0.539/0.551 |
| 917504 | 0.527/0.541/**0.549**/0.556/0.56 |
| 950272 | 0.551/0.559/**0.566**/0.581/0.589 |
| 983040 | 0.564/0.575/**0.586**/0.606/0.613 |
| 1015808 | 0.593/0.602/**0.609**/0.617/0.624 |
| 1048576 | 0.597/0.614/**0.635**/0.643/0.656 |

Figure A.2: Minimum, maximum and quartiles for the large simulation graph in figure 10.3 using a multiplier = 1

| Simulations | Multiplayer = 2 |
|---|---|
| 65536 | $4.31 \cdot 10^{-2}/4.35 \cdot 10^{-2}/\mathbf{4.37 \cdot 10^{-2}}/4.41 \cdot 10^{-2}/4.48 \cdot 10^{-2}$ |
| 131072 | $8.23 \cdot 10^{-2}/8.54 \cdot 10^{-2}/\mathbf{8.7 \cdot 10^{-2}}/8.8 \cdot 10^{-2}/8.82 \cdot 10^{-2}$ |
| 196608 | 0.114/0.118/**0.12**/0.121/0.122 |
| 262144 | 0.159/0.16/**0.162**/0.164/0.168 |
| 327680 | 0.189/0.195/**0.197**/0.2/0.203 |
| 393216 | 0.23/0.235/**0.238**/0.244/0.247 |
| 458752 | 0.27/0.271/**0.274**/0.275/0.28 |
| 524288 | 0.307/0.312/**0.314**/0.316/0.32 |
| 589824 | 0.351/0.355/**0.358**/0.369/0.373 |
| 655360 | 0.381/0.389/**0.396**/0.4/0.401 |
| 720896 | 0.421/0.435/**0.439**/0.443/0.455 |
| 786432 | 0.46/0.468/**0.472**/0.479/0.491 |
| 851968 | 0.511/0.512/**0.514**/0.517/0.527 |
| 917504 | 0.534/0.538/**0.555**/0.566/0.569 |
| 983040 | 0.567/0.588/**0.593**/0.598/0.614 |
| 1048576 | 0.611/0.624/**0.635**/0.647/0.655 |
| 1114112 | 0.65/0.651/**0.656**/0.669/0.672 |
| 1179648 | 0.681/0.708/**0.714**/0.722/0.731 |
| 1245184 | 0.729/0.733/**0.741**/0.749/0.758 |
| 1310720 | 0.782/0.789/**0.793**/0.801/0.806 |
| 1376256 | 0.782/0.812/**0.818**/0.846/0.851 |
| 1441792 | 0.841/0.855/**0.859**/0.864/0.883 |
| 1507328 | 0.861/0.895/**0.914**/0.927/0.945 |
| 1572864 | 0.911/0.94/**0.952**/0.962/0.973 |
| 1638400 | 0.965/0.972/**0.983**/0.987/1.01 |
| 1703936 | 1.02/1.02/**1.03**/1.04/1.05 |
| 1769472 | 1.05/1.06/**1.07**/1.08/1.09 |
| 1835008 | 1.06/1.09/**1.09**/1.11/1.12 |
| 1900544 | 1.11/1.13/**1.14**/1.15/1.18 |
| 1966080 | 1.15/1.15/**1.17**/1.19/1.2 |
| 2031616 | 1.18/1.2/**1.22**/1.23/1.25 |
| 2097152 | 1.23/1.24/**1.25**/1.28/1.29 |

Figure A.3: Minimum, maximum and quartiles for the large simulation graph in figure 10.3 using a multiplier = 2

| Simulations | Multiplayer = 4 |
|---|---|
| 131072 | $8.51 \cdot 10^{-2}/8.6 \cdot 10^{-2}/\mathbf{8.74 \cdot 10^{-2}}/8.76 \cdot 10^{-2}/8.77 \cdot 10^{-2}$ |
| 262144 | 0.17/0.172/**0.172**/0.174/0.174 |
| 393216 | 0.237/0.238/**0.242**/0.245/0.246 |
| 524288 | 0.317/0.324/**0.327**/0.329/0.333 |
| 655360 | 0.393/0.394/**0.397**/0.399/0.405 |
| 786432 | 0.471/0.473/**0.478**/0.484/0.489 |
| 917504 | 0.539/0.544/**0.548**/0.557/0.56 |
| 1048576 | 0.622/0.626/**0.637**/0.642/0.65 |
| 1179648 | 0.706/0.716/**0.722**/0.732/0.739 |
| 1310720 | 0.772/0.782/**0.79**/0.793/0.798 |
| 1441792 | 0.863/0.865/**0.876**/0.888/0.889 |
| 1572864 | 0.927/0.936/**0.94**/0.944/0.965 |
| 1703936 | 1.01/1.02/**1.03**/1.04/1.05 |
| 1835008 | 1.08/1.09/**1.1**/1.11/1.12 |
| 1966080 | 1.16/1.18/**1.19**/1.2/1.2 |
| 2097152 | 1.25/1.26/**1.27**/1.28/1.29 |
| 2228224 | 1.3/1.31/**1.33**/1.36/1.38 |
| 2359296 | 1.37/1.41/**1.43**/1.44/1.46 |
| 2490368 | 1.43/1.48/**1.5**/1.51/1.51 |
| 2621440 | 1.54/1.56/**1.58**/1.59/1.6 |
| 2752512 | 1.62/1.63/**1.65**/1.66/1.68 |
| 2883584 | 1.71/1.71/**1.74**/1.75/1.78 |
| 3014656 | 1.78/1.79/**1.83**/1.85/1.85 |
| 3145728 | 1.85/1.87/**1.88**/1.89/1.91 |
| 3276800 | 1.9/1.95/**1.97**/1.98/2.02 |
| 3407872 | 2.0/2.01/**2.04**/2.07/2.09 |
| 3538944 | 2.06/2.11/**2.11**/2.13/2.16 |
| 3670016 | 2.13/2.16/**2.2**/2.22/2.24 |
| 3801088 | 2.21/2.24/**2.27**/2.28/2.35 |
| 3932160 | 2.31/2.35/**2.36**/2.38/2.4 |
| 4063232 | 2.41/2.42/**2.44**/2.45/2.46 |
| 4194304 | 2.46/2.47/**2.51**/2.55/2.56 |

Figure A.4: Minimum, maximum and quartiles for the large simulation graph in figure 10.3 using a multiplier = 4

| Simulations | Multiplayer = 8 |
|---|---|
| 262144 | 0.173/0.173/**0.174**/0.176/0.178 |
| 524288 | 0.337/0.34/**0.343**/0.344/0.345 |
| 786432 | 0.474/0.479/**0.484**/0.489/0.495 |
| 1048576 | 0.638/0.65/**0.652**/0.655/0.659 |
| 1310720 | 0.775/0.787/**0.791**/0.794/0.801 |
| 1572864 | 0.946/0.955/**0.964**/0.967/0.969 |
| 1835008 | 1.08/1.08/**1.1**/1.11/1.13 |
| 2097152 | 1.24/1.27/**1.28**/1.28/1.29 |
| 2359296 | 1.42/1.43/**1.44**/1.45/1.46 |
| 2621440 | 1.55/1.56/**1.59**/1.59/1.6 |
| 2883584 | 1.69/1.73/**1.74**/1.76/1.77 |
| 3145728 | 1.87/1.88/**1.89**/1.9/1.91 |
| 3407872 | 2.04/2.05/**2.06**/2.07/2.09 |
| 3670016 | 2.17/2.18/**2.19**/2.21/2.25 |
| 3932160 | 2.33/2.35/**2.36**/2.4/2.4 |
| 4194304 | 2.5/2.52/**2.54**/2.54/2.56 |
| 4456448 | 2.63/2.66/**2.67**/2.7/2.72 |
| 4718592 | 2.78/2.81/**2.84**/2.86/2.87 |
| 4980736 | 2.94/2.97/**2.98**/2.99/3.03 |
| 5242880 | 3.12/3.14/**3.16**/3.19/3.2 |
| 5505024 | 3.26/3.27/**3.3**/3.35/3.37 |
| 5767168 | 3.33/3.44/**3.45**/3.47/3.53 |
| 6029312 | 3.55/3.59/**3.62**/3.66/3.68 |
| 6291456 | 3.72/3.76/**3.77**/3.79/3.8 |
| 6553600 | 3.84/3.87/**3.91**/3.96/3.99 |
| 6815744 | 4.03/4.05/**4.1**/4.11/4.13 |
| 7077888 | 4.2/4.22/**4.27**/4.27/4.31 |
| 7340032 | 4.33/4.36/**4.39**/4.42/4.47 |
| 7602176 | 4.44/4.55/**4.58**/4.59/4.66 |
| 7864320 | 4.65/4.67/**4.71**/4.74/4.83 |
| 8126464 | 4.73/4.83/**4.88**/4.9/4.99 |
| 8388608 | 4.94/5.01/**5.04**/5.07/5.09 |

Figure A.5: Minimum, maximum and quartiles for the large simulation graph in figure 10.3 using a multiplier = 8

| Simulations | Results NOT combined | Results combined |
|---|---|---|
| 1K | $2.38{\cdot}10^{-3}/2.66{\cdot}10^{-3}/\mathbf{2.76{\cdot}10^{-3}}/2.83{\cdot}10^{-3}/3.03{\cdot}10^{-3}$ | $2.6{\cdot}10^{-3}/2.78{\cdot}10^{-3}/\mathbf{2.87{\cdot}10^{-3}}/2.93{\cdot}10^{-3}/3.78{\cdot}10^{-3}$ |
| 98K | $5.17{\cdot}10^{-2}/5.52{\cdot}10^{-2}/\mathbf{5.69{\cdot}10^{-2}}/5.85{\cdot}10^{-2}/6.24{\cdot}10^{-2}$ | $6.14{\cdot}10^{-2}/6.82{\cdot}10^{-2}/\mathbf{6.98{\cdot}10^{-2}}/7.15{\cdot}10^{-2}/7.67{\cdot}10^{-2}$ |
| 196K | $9.83{\cdot}10^{-2}$/0.106/**0.109**/0.111/0.118 | 0.121/0.133/**0.137**/0.141/0.15 |
| 293K | 0.141/0.159/**0.163**/0.167/0.176 | 0.188/0.2/**0.206**/0.211/0.223 |
| 390K | 0.193/0.211/**0.216**/0.221/0.239 | 0.24/0.264/**0.271**/0.28/0.294 |
| 487K | 0.232/0.262/**0.268**/0.274/0.292 | 0.29/0.328/**0.337**/0.35/0.369 |
| 585K | 0.271/0.313/**0.323**/0.334/0.35 | 0.371/0.398/**0.408**/0.418/0.451 |
| 682K | 0.327/0.362/**0.374**/0.385/0.409 | 0.423/0.466/**0.476**/0.486/0.516 |
| 779K | 0.393/0.416/**0.431**/0.443/0.465 | 0.494/0.526/**0.544**/0.558/0.591 |
| 877K | 0.447/0.47/**0.484**/0.495/0.515 | 0.537/0.593/**0.61**/0.621/0.68 |
| 974K | 0.444/0.517/**0.537**/0.55/0.576 | 0.59/0.659/**0.677**/0.693/0.73 |

Figure A.6: Minimum, maximum and quartiles for the overhead graph in figure 10.4

| Lookup time | brownian + brownian | brownian |
|---|---|---|
| 0.0 | 0.388/0.405/**0.418**/0.42/0.422 | 0.388/0.396/**0.41**/0.419/0.426 |
| 4.17e-2 | 0.404/0.411/**0.433**/0.447/0.465 | 0.393/0.406/**0.417**/0.435/0.44 |
| 8.33e-2 | 0.438/0.45/**0.461**/0.468/0.472 | 0.408/0.424/**0.437**/0.447/0.456 |
| 0.125 | 0.432/0.479/**0.491**/0.505/0.509 | 0.411/0.434/**0.448**/0.465/0.468 |
| 0.1667 | 0.493/0.496/**0.505**/0.522/0.539 | 0.427/0.442/**0.45**/0.469/0.483 |
| 0.2083 | 0.505/0.521/**0.535**/0.55/0.554 | 0.459/0.467/**0.477**/0.486/0.494 |
| 0.25 | 0.526/0.553/**0.559**/0.569/0.603 | 0.466/0.471/**0.484**/0.496/0.511 |
| 0.2917 | 0.58/0.583/**0.589**/0.606/0.616 | 0.476/0.481/**0.496**/0.507/0.518 |
| 0.3333 | 0.565/0.597/**0.609**/0.62/0.622 | 0.485/0.494/**0.517**/0.523/0.541 |
| 0.375 | 0.611/0.623/**0.636**/0.643/0.656 | 0.498/0.507/**0.519**/0.532/0.546 |
| 0.4167 | 0.644/0.65/**0.659**/0.676/0.678 | 0.511/0.524/**0.528**/0.539/0.552 |
| 0.4583 | 0.672/0.68/**0.692**/0.701/0.705 | 0.522/0.535/**0.552**/0.564/0.576 |
| 0.5 | 0.693/0.711/**0.726**/0.736/0.75 | 0.546/0.554/**0.557**/0.566/0.588 |
| 0.5417 | 0.716/0.739/**0.742**/0.754/0.76 | 0.554/0.57/**0.577**/0.583/0.599 |
| 0.5833 | 0.731/0.752/**0.767**/0.78/0.789 | 0.567/0.579/**0.586**/0.6/0.615 |
| 0.625 | 0.738/0.782/**0.795**/0.799/0.811 | 0.564/0.587/**0.591**/0.607/0.625 |
| 0.6667 | 0.787/0.811/**0.816**/0.826/0.839 | 0.584/0.6/**0.612**/0.627/0.638 |
| 0.7083 | 0.814/0.84/**0.848**/0.859/0.868 | 0.61/0.625/**0.635**/0.644/0.651 |
| 0.75 | 0.839/0.86/**0.871**/0.881/0.889 | 0.6/0.625/**0.633**/0.645/0.666 |
| 0.7917 | 0.87/0.877/**0.89**/0.909/0.93 | 0.61/0.646/**0.656**/0.669/0.679 |
| 0.8333 | 0.889/0.909/**0.925**/0.933/0.94 | 0.618/0.644/**0.659**/0.671/0.692 |
| 0.875 | 0.929/0.936/**0.957**/0.964/0.97 | 0.654/0.662/**0.676**/0.682/0.695 |
| 0.9167 | 0.948/0.953/**0.976**/0.979/0.987 | 0.658/0.687/**0.697**/0.7/0.704 |
| 0.9583 | 0.982/0.992/**0.997**/1.01/1.02 | 0.694/0.698/**0.704**/0.723/0.73 |
| 1.0 | 0.988/1.01/**1.02**/1.03/1.07 | 0.695/0.716/**0.723**/0.734/0.741 |

Figure A.7: Minimum, maximum and quartiles for the de-nesting graph in figure 10.5

| Lookup time | maximum brownian |
|---|---|
| 0.0 | 0.702/0.708/**0.721**/0.738/0.755 |
| 4.17e-2 | 0.697/0.707/**0.725**/0.739/0.756 |
| 8.33e-2 | 0.694/0.702/**0.72**/0.724/0.731 |
| 0.125 | 0.689/0.705/**0.715**/0.718/0.75 |
| 0.1667 | 0.702/0.705/**0.718**/0.741/0.746 |
| 0.2083 | 0.684/0.701/**0.713**/0.724/0.728 |
| 0.25 | 0.686/0.707/**0.718**/0.733/0.737 |
| 0.2917 | 0.68/0.701/**0.719**/0.736/0.753 |
| 0.3333 | 0.714/0.722/**0.727**/0.739/0.748 |
| 0.375 | 0.706/0.711/**0.727**/0.732/0.744 |
| 0.4167 | 0.706/0.712/**0.717**/0.728/0.736 |
| 0.4583 | 0.709/0.724/**0.725**/0.737/0.753 |
| 0.5 | 0.709/0.72/**0.727**/0.738/0.752 |
| 0.5417 | 0.712/0.713/**0.717**/0.731/0.763 |
| 0.5833 | 0.706/0.718/**0.727**/0.738/0.756 |
| 0.625 | 0.687/0.712/**0.725**/0.741/0.752 |
| 0.6667 | 0.705/0.711/**0.735**/0.74/0.744 |
| 0.7083 | 0.706/0.716/**0.734**/0.749/0.772 |
| 0.75 | 0.711/0.718/**0.731**/0.746/0.761 |
| 0.7917 | 0.71/0.718/**0.73**/0.735/0.762 |
| 0.8333 | 0.714/0.724/**0.732**/0.752/0.759 |
| 0.875 | 0.701/0.715/**0.733**/0.746/0.766 |
| 0.9167 | 0.708/0.729/**0.742**/0.747/0.759 |
| 0.9583 | 0.709/0.732/**0.738**/0.745/0.753 |
| 1.0 | 0.691/0.734/**0.743**/0.748/0.759 |

Figure A.8: Minimum, maximum and quartiles for the skip graph in figure 10.6

# Appendix B

# Selected SPL modules

## B.1 Module Language.SPL

```
{-# LANGUAGE GADTs, KindSignatures, FlexibleInstances, FlexibleContexts,
     MultiParamTypeClasses, NoMonomorphismRestriction, TypeFamilies #-}

module Language.SPL (
    -- * Built-in constructs
    -- ** Distributions
    uniform, normal, lookup, sample,
    -- ** Processes
    trace, closed, prefix, zip, skip,
    -- * Prelude
    inclusivePrefix,
    iterative, map, time, always, reverse,
    brownian, integral,
    true, false,
    zip3,
    curry, uncurry, curry3, uncurry3,
    lift, lift2, lift3,
    choose, choice,
    fold, maximum_, minimum_, average,
    -- * Types
    Pair (..),
    If_ (..),
    Dist, Process,
    Time,
    Type,
    Ordered (..),
    Boolean (..),
    ToConstant (..)
    ) where

import Language.SPL.Syntax

import Prelude hiding (curry, uncurry, zip, zip3, map, lookup, reverse)
import qualified Prelude as H
import Data.Char (toLower)


-------------------- Built-in constructs --------------------
```

```
-- |The standard uniform distribution (between 0 and 1, both inclusive).
uniform :: Dist Double
uniform = Uniform

-- |The standard normal distribution (with mean 0 and variance 1).
normal :: Dist Double
normal = Normal

-- |Applies the function to a single sample from the distribution.
--  The sample is a so called degenerate distribution containing one value with
     probability 1.
sample :: (Type a, Type b) => Dist a -> (Dist a -> Dist b) -> Dist b
sample = Sample

-- |A Process is conceptually a function from time to Dist.
--  Lookup is the manifistation of this concept.
lookup :: Type a => Dist Time -> Process a -> Dist a
lookup = Lookup


-- |Builds a process given function from time to a distribution
closed :: Type a => (Dist Time -> Dist a) -> Process a
closed = Closed

-- |Builds a process by accumulating over another process, given an initial value.
--  It is similar to a scan over a list (or a prefix sum) - however, it also knows
--  the delta time, specifying the time between the current and previous iteration.
--  The distribution at time zero is the function applied to the delta time, the
--  initial distribution and the distribution at time zero in the other process.
prefix :: (Type a, Type b) => (Dist Time -> Dist a -> Dist b -> Dist a) -> Dist a ->
      Process b -> Process a
prefix = Prefix

-- |Builds a process by pairing each element of two processes.
zip :: (Type a, Type b) => Process a -> Process b -> Process (a, b)
zip = Zip

-- |Applies the function to a single path of the process.
--  The path is a time series (every distribution in the process is degenerate).
trace :: (Type a, Type b) => Process a -> (Process a -> Process b) -> Process b
trace = Trace

-- |Skips ahead in the given process by the specified duration.
skip :: Dist Time -> Process a -> Process a
skip t process = case process of
    Closed f -> Closed (\t' -> f (t + t'))
    Zip p1 p2 -> Zip (skip t p1) (skip t p2)
    TagP tag p -> TagP tag (skip t p)
    Prefix f d0 p -> Prefix f (Lookup t (inclusivePrefix f d0 p)) (skip t p)
    Trace p f -> Trace p (skip t . f)


-------------------- Prelude --------------------


-- |Like prefix, but the whole process is delayed by the delta time
--  and the distribution at time zero is the given initial distribution.
inclusivePrefix :: (Type a, Type b) => (Dist Time -> Dist a -> Dist b -> Dist a) ->
     Dist a -> Process b -> Process a
inclusivePrefix f v p = map first (prefix (\d a w -> pair (second a) (f d (second a)
      w)) (pair v v) p)
```

```
integral = inclusivePrefix (\dt a v -> a + dt * v) 0

brownian = iterative (\dt a -> a + sqrt dt * normal) 0


time = closed id


true = constant True

false = constant False


always = Closed . const


uncurry :: (Type a, Type b, Type c) =>
    (Dist a -> Dist b -> Dist c) -> Dist (a, b) -> Dist c
uncurry f v = f (first v) (second v)

curry :: (Type a, Type b, Type c) =>
    (Dist (a, b) -> Dist c) -> Dist a -> Dist b -> Dist c
curry f a b = f (pair a b)


uncurry3 :: (Type a, Type b, Type c, Type d) =>
    (Dist a -> Dist b -> Dist c -> Dist d) -> Dist (a, (b, c)) -> Dist d
uncurry3 f v = uncurry (f (first v)) (second v)

curry3 :: (Type a, Type b, Type c, Type d) =>
    (Dist (a, (b, c)) -> Dist d) -> Dist a -> Dist b -> Dist c -> Dist d
curry3 f a b c = f (pair a (pair b c))


zip3 a b c = zip a (zip b c)


lift :: (Type a, Type b) =>
    (Dist a -> Dist b) -> Process a -> Process b
lift = map

lift2 :: (Type a, Type b, Type c) =>
    (Dist a -> Dist b -> Dist c) -> Process a -> Process b -> Process c
lift2 f p1 p2 = map (uncurry f) (zip p1 p2)

lift3 :: (Type a, Type b, Type c, Type d) =>
    (Dist a -> Dist b -> Dist c -> Dist d) -> Process a -> Process b -> Process c ->
          Process d
lift3 f p1 p2 p3 = map (uncurry3 f) (zip3 p1 p2 p3)


iterative :: (Type a) =>
    (Dist Time -> Dist a -> Dist a) -> Dist a -> Process a
iterative f i = inclusivePrefix (\dt a _ -> f dt a) i time

map :: (Type a, Type b) =>
    (Dist a -> Dist b) -> Process a -> Process b
map f = prefix (\_ _ d -> f d) (error "Accumulator variable should not be used in a
     map")

reverse :: (Type a) =>
```

```
    Dist Time -> Process a -> Process a
reverse end p = closed $ \t -> lookup (end - t) p


choice q d1 d2 = if_ (uniform .<. q) d1 d2

choose [d] = d
choose (d:ds) = choice (1 / fromIntegral (length (d:ds))) d (choose ds)

fold :: (Type a, Type b) => Dist Time -> Dist Time -> (Dist Time -> Dist a -> Dist b
      -> Dist a) -> (Dist a) -> Process b -> Dist a
fold t1 t2 op init p = lookup (t2 - t1) (prefix op init (skip t1 p))

maximum_ :: Dist Time -> Dist Time -> Process Double -> Dist Double
maximum_ t1 t2 p = fold t1 t2 (const max_) (-1/0) p

minimum_ :: Dist Time -> Dist Time -> Process Double -> Dist Double
minimum_ t1 t2 p = fold t1 t2 (const min_) (1/0) p

average :: Process Double -> Process Double
average process =
    let sumCount = prefix (\_ acc value -> pair (first acc + value) (second acc + 1)
        ) (pair 0 0) process in
    map (\p -> first p / second p) sumCount


-------------------- Constants --------------------


class ToConstant a where
    type ConstantType a
    constant :: ConstantType a -> a

instance ToConstant (Dist Double) where
    type ConstantType (Dist Double) = Double
    constant = Certain . Double

instance ToConstant (Dist Bool) where
    type ConstantType (Dist Bool) = Bool
    constant = Certain . Bool

instance ToConstant (Process Double) where
    type ConstantType (Process Double) = Double
    constant = always . constant

instance ToConstant (Process Bool) where
    type ConstantType (Process Bool) = Bool
    constant = always . constant


-------------------- Syntactic sugar for pairs --------------------


class Pair a where
    pair :: (Type b, Type c) => a b -> a c -> a (b, c)
    first :: (Type b, Type c) => a (b, c) -> a b
    second :: (Type b, Type c) => a (b, c) -> a c

instance Pair Dist where
    pair = Binary Pair
    first = Unary First
    second = Unary Second
```

```
instance Pair Process where
    pair a b = zip a b
    first = lift first
    second = lift second


-------------------- Conditional operator --------------------


class If_ a where
    if_ :: Type b => a Bool -> a b -> a b -> a b

instance If_ Dist where
    if_ = Ternary If

instance If_ Process where
    if_ = lift3 if_


-------------------- Ord emulation --------------------


infix 4 .<.
infix 4 .<=.
infix 4 .>.
infix 4 .>=.
infix 4 .==.
infix 4 ./=.

class Type b => Ordered a b where
    min_ :: a b -> a b -> a b
    max_ :: a b -> a b -> a b
    (.<.) :: a b -> a b -> a Bool
    (.<=.) :: a b -> a b -> a Bool
    (.>.) :: a b -> a b -> a Bool
    (.>=.) :: a b -> a b -> a Bool
    (.==.) :: a b -> a b -> a Bool
    (./=.) :: a b -> a b -> a Bool

infixr 3 .&&.
infixr 2 .||.

class Boolean a where
    (.||.) :: a Bool -> a Bool -> a Bool
    (.&&.) :: a Bool -> a Bool -> a Bool
    not_ :: a Bool -> a Bool


-------------------- Instances for Double --------------------


instance Eq (Dist Double) where
    (==) = error "Eq is not defined for Dist Double, but is still instantiated to
         provide Num (Dist Double)"

instance Show (Dist a) where
    show = showDist 0

instance Num (Dist Double) where
    fromInteger = constant . fromIntegral
    (+) = Binary Add
```

```
    (-) = Binary Sub
    (*) = Binary Mult
    abs = Unary Abs
    signum = Unary Sign
    negate = Unary Negate

instance Fractional (Dist Double) where
    fromRational = constant . fromRational
    (/) = Binary Div
    recip = (1.0 /)

instance Floating (Dist Double) where
    exp = Unary Exp
    pi = constant pi
    log = Unary Log
    sqrt = Unary Sqrt
    sin = Unary Sin
    cos = Unary Cos
    tan = Unary Tan
    asin = Unary Asin
    acos = Unary Acos
    atan = Unary Atan
    sinh = Unary Sinh
    cosh = Unary Cosh
    tanh = Unary Tanh
    asinh = Unary Asinh
    acosh = Unary Acosh
    atanh = Unary Atanh
    (**) = Binary Power
    logBase = Binary LogBase

instance Ordered Dist Double where
    min_ = Binary Min
    max_ = Binary Max
    (.<.) = Binary Less
    (.<=.) = Binary LessEqual
    (.>.) = Binary Greater
    (.>=.) = Binary GreaterEqual
    (.==.) = Binary Equal
    (./=.) = Binary NotEqual

instance Boolean Dist where
    (.||.) = Binary Or
    (.&&.) = Binary And
    not_ = Unary Not

instance Eq (Process Double) where
    (==) = error "Eq is not defined for Process Double, but is still instantiated to
          provide Num (Process Double)"

instance Show (Process a) where
    show = showProcess 0

instance Num (Process Double) where
    fromInteger = constant . fromInteger
    (+) = lift2 (+)
    (-) = lift2 (-)
    (*) = lift2 (*)
    abs = lift abs
    signum = lift signum
    negate = lift negate
```

```
instance Fractional (Process Double) where
    fromRational = constant . fromRational
    (/) = lift2 (/)
    recip = (1.0 /)

instance Floating (Process Double) where
    exp = lift exp
    pi = constant pi
    log = lift log
    sqrt = lift sqrt
    sin = lift sin
    cos = lift cos
    tan = lift tan
    asin = lift asin
    acos = lift acos
    atan = lift atan
    sinh = lift sinh
    cosh = lift cosh
    tanh = lift tanh
    asinh = lift asinh
    acosh = lift acosh
    atanh = lift atanh
    (**) = lift2 (**)
    logBase = lift2 logBase

instance Ordered Process Double where
    min_ = lift2 min_
    max_ = lift2 max_
    (.<.) = lift2 (.<.)
    (.<=.) = lift2 (.<=.)
    (.>.) = lift2 (.>.)
    (.>=.) = lift2 (.>=.)
    (.==.) = lift2 (.==.)
    (./=.) = lift2 (./=.)

instance Boolean Process where
    (.||.) = lift2 (.||.)
    (.&&.) = lift2 (.&&.)
    not_ = lift not_


-------------------- Auxiliaries --------------------


showDist :: Int -> Dist a -> String
showDist x dist = case dist of
    Certain value -> case value of
        Double value -> show value
        Bool value -> H.map toLower (show value)
    Unary op d1 -> parenthesize $ show op ++ " " ++ showDist x d1
    Binary op d1 d2 -> parenthesize $ showDist x d1 ++ " " ++ show op ++ " " ++
         showDist x d2
    Ternary If d1 d2 d3 -> parenthesize $ "if " ++ showDist x d1 ++ " " ++ showDist
         x d2 ++ " " ++ showDist x d3
    Uniform -> "uniform"
    Normal -> "normal"
    Lookup time process -> parenthesize $ "lookup " ++ showDist x time ++ " " ++
         show process
    Sample d f -> parenthesize $ showDist x d ++ " `sample` " ++ showDistFunction x
         f
    TagD i -> showDistVar i
```

```
showDistVar :: Int -> String
showDistVar x = "d" ++ show x

showDistArgument :: Int -> Dist a -> String
showDistArgument x d | varUsedInDist x d = showDistVar x
showDistArgument x _ = "_"

showDistFunction :: Type a => Int -> (Dist a -> Dist b) -> String
showDistFunction x f = "\\" ++ showDistVar x ++ " -> " ++ showDist (x + 1) (f (TagD
    x))

showProcess :: Int -> Process a -> String
showProcess x process = case process of
    Closed f | varUsedInDist x (f (TagD x)) -> parenthesize $ "closed \\" ++
        showDistVar x ++ " -> " ++ showDist (x + 1) (f (TagD x))
            | otherwise -> showDist x (f (TagD undefined))
    Prefix f d0 p ->
        let body = f (TagD x) (TagD (x + 1)) (TagD (x + 2)) in
        let a1 = showDistArgument x body in
        let a2 = showDistArgument (x + 1) body in
        let a3 = showDistArgument (x + 2) body in
        let d0' = if varUsedInDist (x + 1) body then showDist x d0 else "_" in
        let p' = if varUsedInDist (x + 2) body then showProcess x p else "_" in
        parenthesize $ "prefix " ++ lambda [a1, a2, a3] (showDist (x + 3) body) ++ "
            " ++ d0' ++ " " ++ p'
    Zip p1 p2 -> parenthesize $ "zip " ++ showProcess x p1 ++ " " ++ showProcess x
        p2
    Trace p f ->
        let body = f (TagP x undefined) in
        let a1 = showProcessArgument x body in
        parenthesize $ showProcess x p ++ " `trace` " ++ lambda [a1] (showProcess (x
            + 1) body)
    TagP x _ -> showProcessVar x

showProcessVar :: Int -> String
showProcessVar x = "p" ++ show x

showProcessArgument :: Int -> Process a -> String
showProcessArgument x p | varUsedInProcess x p = showProcessVar x
showProcessArgument x _ = "_"

lambda :: [String] -> String -> String
lambda args body = parenthesize $ "\\" ++ (concatMap (++ " ") args) ++ "-> " ++
    unparenthesize body

parenthesize s = "(" ++ s ++ ")"

unparenthesize s =
    if length s >= 2 && head s == '(' && s !! (length s - 1) == ')'
    then take (length s - 2) (tail s)
    else s
```

## B.2 Module Language.SPL.Syntax

```
{-# LANGUAGE GADTs, KindSignatures, TypeFamilies, OverlappingInstances,
    EmptyDataDecls, DeriveDataTypeable, StandaloneDeriving, FlexibleInstances,
    ScopedTypeVariables #-}

module Language.SPL.Syntax (
    module Language.SPL.Operator,
    module Language.SPL.Type,
```

```
    Time,
    Dist (..),
    Process (..),
    Type (..),
    varUsedInDist,
    varUsedInProcess,
    examine,
    usesAccumulator
) where

import Language.SPL.Operator
import Language.SPL.Type

import Data.Typeable
import System.Random


type Time = Double


data Dist :: * -> * where
    Normal :: Dist Double
    Uniform :: Dist Double
    Lookup :: (Type a) => Dist Time -> Process a -> Dist a
    Certain :: (Type a) => Constant a -> Dist a
    Sample :: (Type a, Type b) => Dist a -> (Dist a -> Dist b) -> Dist b
    Unary :: (Type a1, Type a2) => UnaryOperator a1 a2 -> Dist a1 -> Dist a2
    Binary :: (Type a1, Type a2, Type a3) => BinaryOperator a1 a2 a3 -> Dist a1 ->
        Dist a2 -> Dist a3
    Ternary :: (Type a1, Type a2, Type a3, Type a4) => TernaryOperator a1 a2 a3 a4
        -> Dist a1 -> Dist a2 -> Dist a3 -> Dist a4
    TagD :: (Type a) => Int -> Dist a


data Process :: * -> * where
    Closed :: (Type a) => (Dist Time -> Dist a) -> Process a
    Prefix :: (Type a, Type b) => (Dist Time -> Dist a -> Dist b -> Dist a) -> Dist
        a -> Process b -> Process a
    Zip :: (Type a, Type b) => Process a -> Process b -> Process (a, b)
    Trace :: (Type a, Type b) => Process a -> (Process a -> Process b) -> Process b
    TagP :: (Type a) => Int -> Process a -> Process a


-- | Contains the types that we can represent (in 'Dist' and 'Process').
class Typeable a => Type a where
    splType :: a -> SPLType
    toDist :: a -> Dist a

instance Type Double where
    splType _ = DoubleType
    toDist = Certain . Double

instance Type Bool where
    splType _ = BoolType
    toDist = Certain . Bool

deriving instance Typeable StdGen

instance Type StdGen where
    splType _ = GeneratorType
    toDist = error "Unimplemented" -- TODO: Fix this (although it cannot happen if
        we don't expose toDist)
```

```
instance forall a b. (Type a, Type b) => Type (a, b) where
    splType _ = PairType (splType (error "splType" :: a)) (splType (error "splType"
        :: b))
    toDist (a, b) = Binary Pair (toDist a) (toDist b)


varUsedInDist :: Int -> Dist a -> Bool
varUsedInDist x dist = case dist of
    TagD i -> i == x
    Unary _ d1 -> varUsedInDist x d1
    Binary _ d1 d2 -> varUsedInDist x d1 || varUsedInDist x d2
    Ternary If d1 d2 d3 -> varUsedInDist x d1 || varUsedInDist x d2 || varUsedInDist
         x d3
    Lookup time process -> varUsedInDist x time || varUsedInProcess x process
    Sample d f -> varUsedInDist x d || varUsedInDist x (f (TagD (x+1)))
    _ -> False

varUsedInProcess :: Int -> Process a -> Bool
varUsedInProcess x process = case process of
    TagP i _ -> x == i
    Closed f -> varUsedInDist x (f (TagD (x+1)))
    Trace p f -> varUsedInProcess x p || varUsedInProcess x (f (TagP (x+1) undefined
        ))
    Prefix f d0 p ->
        let body = f (TagD (x+1)) (TagD (x+2)) (TagD (x+3)) in
        let usingAccumulator = varUsedInDist (x + 2) body in
        let usedInAccumulator = usingAccumulator && varUsedInDist x d0 in
        varUsedInDist x body || usedInAccumulator || varUsedInProcess x p
    Zip p1 p2 -> varUsedInProcess x p1 || varUsedInProcess x p2

examine :: (Type a1, Type a2, Type a) =>
    (Dist a -> Dist a1 -> Dist a2 -> Dist a3) -> (Bool, Bool, Bool)
examine f =
    let (x1, x2, x3) = (-4, -3, -2) in
    let body = f (TagD x1) (TagD x2) (TagD x3) in
    (varUsedInDist x1 body, varUsedInDist x2 body, varUsedInDist x3 body)

usesAccumulator f = let (_, used, _) = examine f in used
```

## B.3 Module Language.SPL.Semantics

```
{-# LANGUAGE GADTs #-}

module Language.SPL.Semantics where

import Language.SPL.Syntax
import Language.SPL.Operator
import Language.SPL ()
import Prelude hiding (Real)

import Control.Monad


type Real = Double


delta = 0.1
end = 10
```

```
class Monad m => ProbabilityMonad m where
    uniform' :: m Real
    normal'  :: m Real


process :: ProbabilityMonad m => Process a -> m [a]
process p = case p of
    Closed f -> mapM (distribution . f . Certain . Double) [0, delta .. end]
    Prefix f i p | usesAccumulator f -> do
        i' <- distribution i
        p' <- process p
        let accumulate a v = distribution (f (toDist delta) (toDist a) (toDist v))
        l <- scanM accumulate i' p'
        return (tail l)
    Prefix f i p -> do
        p' <- process p
        mapM (distribution . f (toDist delta) undefined . toDist) p'
    Zip p1 p2 -> do
        p1' <- process p1
        p2' <- process p2
        return (zip p1' p2')
    Trace p f -> do
        p' <- process p
        let s = Closed (\(Certain (Double t')) -> toDist (index t' p'))
        process (f s)


distribution :: ProbabilityMonad m => Dist a -> m a
distribution d = case d of
    Uniform -> uniform'
    Normal -> normal'
    Certain (Double v) -> return v
    Certain (Bool v) -> return v
    Lookup t p -> do
        t' <- distribution t
        p' <- process p
        return (index t' p')
    Sample d f -> do
        d' <- distribution d
        distribution (f (toDist d'))
    Unary o d -> do
        d' <- distribution d
        return (unaryOperator o d')
    Binary o d1 d2 -> do
        d1' <- distribution d1
        d2' <- distribution d2
        return (binaryOperator o d1' d2')
    Ternary o d1 d2 d3 -> do
        d1' <- distribution d1
        d2' <- distribution d2
        d3' <- distribution d3
        return (ternaryOperator o d1' d2' d3')


index t l = l !! floor (t / delta)


scanM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m [a]
scanM f i [] = return [i]
scanM f i (x:xs) = do
    i' <- f i x
    is <- scanM f i' xs
```

```
    return (i:is)


{-
processAt :: ProbabilityMonad m => Process a -> Time -> m a
processAt p t = case p of
    Closed f -> distribution (f (Certain (Double t)))
    Prefix f i p | t == 0 -> do
        i' <- distribution i
        p' <- processAt p t
        distribution (f (toDist delta) (toDist i') (toDist p'))
    Prefix f i p -> do
        a <- processAt (Prefix f i p) (t - delta)
        p' <- processAt p t
        distribution (f (toDist delta) (toDist a) (toDist p'))
    Zip p1 p2 -> do
        p1' <- processAt p1 t
        p2' <- processAt p2 t
        return (p1', p2')
    -- The case for Trace is wrong: it mixes up values from different time series
    Trace p f -> do
        l <- sequence [processAt p t' | t' <- [0, delta .. end]]
        let index t' = toDist (l !! floor (t' / delta))
        let s = Closed (\(Certain (Double t')) -> index t')
        processAt (f s) t
-}
```

## B.4 Module Language.SPL.Intermediate

```
{-# LANGUAGE GADTs, KindSignatures, MultiParamTypeClasses, FlexibleContexts,
     FlexibleInstances, ScopedTypeVariables, TypeFamilies, EmptyDataDecls #-}

module Language.SPL.Intermediate (
    Intermediate (..),
    Index (..), Layout (..), peek,
    Translate (..), translate, convert,
    Accumulator (..),
    stochastic
    ) where

import qualified Language.SPL.Syntax as S
import qualified Language.SPL.Operator as S
import qualified Language.SPL as S
import Language.SPL.SimulationResult (SimulationResult)

import System.Random
import Data.Typeable
import Data.Maybe
import Control.Monad (guard)


data Index :: * -> * -> * where
    Zero :: Index (env, a) a
    Succ :: Index env a -> Index (env, b) a

data Intermediate :: * -> * -> * where
    Uniform :: Intermediate env Double
    Normal :: Intermediate env Double
    Let :: (S.Type a, S.Type b) =>
        Intermediate env a -> Intermediate (env, a) b -> Intermediate env b
    Constant :: (S.Type a) =>
```

```
        (S.Constant a) -> Intermediate env a
    Unary :: (S.Type a, S.Type b) =>
        (S.UnaryOperator a b) -> Intermediate env a -> Intermediate env b
    Binary :: (S.Type a, S.Type b, S.Type c) =>
        (S.BinaryOperator a b c) -> Intermediate env a -> Intermediate env b ->
            Intermediate env c
    If :: (S.Type a) =>
        Intermediate env Bool -> Intermediate env a -> Intermediate env a ->
            Intermediate env a
    Prefix :: (S.Type a) =>
        Bool -> Intermediate env S.Time -> Accumulator env a -> Intermediate env a
    Variable :: (S.Type a) =>
        Index env a -> Intermediate env a
    Split :: (S.Type a) =>
        Intermediate (env, StdGen) a -> Intermediate env a
    Use :: (S.Type a) =>
        Index env StdGen -> Intermediate env a -> Intermediate env a

data Accumulator :: * -> * -> * where
    Accumulate :: (S.Type a, S.Type b) =>
        Intermediate (((env, S.Time), a), b) a ->
        Bool -> Maybe (Intermediate env a) -> Maybe (Accumulator env b) ->
            Accumulator env a
    Zip :: (S.Type a, S.Type b) =>
        Accumulator env a -> Accumulator env b -> Accumulator env (a, b)
    Expression :: (S.Type a) =>
        Bool -> Intermediate (env, S.Time) a -> Accumulator env a
    Splitting :: (S.Type a) =>
        Accumulator (env, StdGen) a -> Accumulator env a
    Using :: (S.Type a) =>
        Index env StdGen -> Accumulator env a -> Accumulator env a


data Layout :: * -> * -> * where
    Empty :: Layout env ()
    Push :: (S.Type a) =>
        Layout env env' -> Index env a -> Layout env (env', a)

size :: Layout env env' -> Int
size Empty = 0
size (Push layout _) = size layout + 1

increase :: Layout env env' -> Layout (env, a) env'
increase Empty = Empty
increase (Push layout index) = Push (increase layout) (Succ index)

project :: (S.Type a) => Int -> Layout env env' -> Index env a
project _ Empty = error "Cannot project an empty layout"
project 0 (Push _ index) = fromJust (gcast index)
project n (Push layout _) = project (n - 1) layout

peek :: Index env a -> env -> a
peek Zero (_, a) = a
peek (Succ n) (environment, _) = peek n environment


translate :: Translate () a => a -> Intermediate (EnvironmentOf () a) (ResultOf a)
translate = translate' Empty

class Translate env a where
    type EnvironmentOf env a
    type ResultOf a
```

```
    translate' :: Layout env env -> a -> Intermediate (EnvironmentOf env a) (
        ResultOf a)


instance S.Type a => Translate env (S.Dist a) where
    type EnvironmentOf env (S.Dist a) = env
    type ResultOf (S.Dist a) = a
    translate' = convert'

instance forall env a b. (Translate (env, a) b, S.Type a) => Translate env (S.Dist a
     -> b) where
    type EnvironmentOf env (S.Dist a -> b) = EnvironmentOf (env, a) b
    type ResultOf (S.Dist a -> b) = ResultOf b
    translate' layout f = translate' layout' (f tag)
        where
            (tag, layout') = distTag layout

instance forall env a b. (Translate (env, a) b, S.Type a) => Translate env (S.
    Process a -> b) where
    type EnvironmentOf env (S.Process a -> b) = EnvironmentOf (env, a) b
    type ResultOf (S.Process a -> b) = ResultOf b
    translate' layout f = translate' layout' (f (S.always tag))
        where
            (tag, layout') = distTag layout


convert :: S.Dist a -> Intermediate () a
convert = convert' Empty

convert' :: Layout env env -> S.Dist a -> Intermediate env a
convert' layout term = case term of
    S.TagD tag -> Variable (project (size layout - tag - 1) layout)
    S.Certain constant -> Constant constant
    S.Uniform -> Uniform
    S.Normal -> Normal
    S.Sample e f -> Let (convert' layout e) (convert' layout' (f tag))
        where
            (tag, layout') = distTag layout
    S.Unary op e1 -> Unary op (convert' layout e1)
    S.Binary op e1 e2 -> Binary op (convert' layout e1) (convert' layout e2)
    S.Ternary S.If e1 e2 e3 -> If (convert' layout e1) (convert' layout e2) (convert
        ' layout e3)
    S.Lookup e p -> lookup layout e p

    where

        lookup :: Layout env env -> S.Dist S.Time -> S.Process a -> Intermediate env
             a
        lookup layout time process = case process of
            p@(S.Prefix _ _ p') ->
                Prefix (usesTimeInProcess p') (convert' layout time) (accumulator
                    layout p)
            S.Zip p1 p2 ->
                Binary S.Pair (lookup layout time p1) (lookup layout time p2)
            S.Closed f ->
                convert' layout (f time)
            S.Trace p f ->
                Split (lookup layout' time (f tag))
                where
                    (tag, layout') = processTag layout p
            S.TagP tag p ->
                Use (project (size layout - tag - 1) layout) (lookup layout time p)
```

```
        accumulator :: (S.Type a) => Layout env env -> S.Process a -> Accumulator
             env a
        accumulator layout process = case process of
            S.Zip p1 p2 ->
                let a1 = accumulator layout p1 in
                let a2 = accumulator layout p2 in
                Zip a1 a2
            S.Prefix f e p ->
                let accumulator' = accumulator layout p in
                let initialValue = convert' layout e in

                let (timeTag,       layout')   = distTag layout in
                let (accumulateTag, layout'')  = distTag layout' in
                let (valueTag,      layout''') = distTag layout'' in
                let f' = convert' layout''' (f timeTag accumulateTag valueTag) in

                let (useDt, useAccumulator, useProcess) = S.examine f in

                let justWhen :: Bool -> a -> Maybe a
                    justWhen condition a = guard condition >> return a in

                Accumulate f' useDt (justWhen useAccumulator initialValue) (justWhen
                      useProcess accumulator')
            S.Trace p f -> Splitting (accumulator layout' (f tag))
                where
                    (tag, layout') = processTag layout p
            S.TagP tag p ->
                Using (project (size layout - tag - 1) layout) (accumulator layout p
                     )
            p@(S.Closed _) ->
                let (tag, layout') = distTag layout in
                Expression (usesTimeInProcess p) (lookup layout' tag p)


distTag :: S.Type a =>
    Layout env env' ->
    (S.Dist a, Layout (env, a) (env', a))
distTag layout = (S.TagD (size layout), increase layout `Push` Zero)

processTag :: S.Type a =>
     Layout env env' ->
     S.Process a ->
     (S.Process a, Layout (env, StdGen) (env', StdGen))
processTag layout p = (S.TagP (size layout) p, increase layout `Push` Zero)


usesTimeInProcess :: S.Process a -> Bool
usesTimeInProcess process = case process of
    S.Zip p1 p2 -> usesTimeInProcess p1 || usesTimeInProcess p2
    S.Prefix _ _ p -> usesTimeInProcess p
    S.Trace p f -> usesTimeInProcess (f p)
    S.Closed f -> S.varUsedInDist 0 (f (S.TagD 0))
    S.TagP _ p -> usesTimeInProcess p


stochastic :: Intermediate env a -> Bool
stochastic process = case process of
    Uniform -> True
    Normal -> True
    Let e1 e2 -> stochastic e1 || stochastic e2
    Constant _ -> False
```

```
    Unary _ e1 -> stochastic e1
    Binary _ e1 e2 -> stochastic e1 || stochastic e2
    If e1 e2 e3 -> stochastic e1 || stochastic e2 || stochastic e3
    Prefix _ e a -> stochastic e || stochasticAccumulator a
    Variable _ -> False
    Split e -> stochastic e
    Use _ e -> False

stochasticAccumulator :: Accumulator env a -> Bool
stochasticAccumulator accumulator = case accumulator of
    Accumulate e1 _ e2 a -> stochastic e1 || maybe False stochastic e2 || maybe
        False stochasticAccumulator a
    Zip a1 a2 -> stochasticAccumulator a1 || stochasticAccumulator a2
    Expression _ e -> stochastic e
    Splitting a -> stochasticAccumulator a
    Using _ a -> stochasticAccumulator a


instance Show (Intermediate env a) where
    show intermediate = case intermediate of
        Uniform -> "uniform"
        Normal -> "normal"
        Let e1 e2 -> "(let " ++ show e1 ++ " in " ++ show e2 ++ ")"
        Constant (S.Double value) -> show value
        Constant (S.Bool value) -> show value
        Unary op e1 -> "(" ++ show op ++ " (" ++ show e1 ++ "))"
        Binary op e1 e2 -> "(" ++ show e1 ++ show op ++ show e2 ++ ")"
        If e1 e2 e3 -> "(if " ++ show e1 ++ " then " ++ show e2 ++ " else " ++ show
            e3 ++ ")"
        Variable index -> "x" ++ show index
        Split e -> "(split (" ++ show e ++ "))"
        Use index e -> "(use g" ++ show index ++ " within " ++ show e ++ ")"
        Prefix usingTime e a -> "(prefix " ++ show usingTime ++ " " ++ show e ++ " "
             ++ show a ++ ")"

instance Show (Accumulator env a) where
    show accumulator = case accumulator of
        Accumulate f useDt d0 acc -> "(accumulate " ++ show f ++ " " ++ show useDt
            ++ " " ++ show d0 ++ " " ++ show acc ++ ")"
        Zip acc1 acc2 -> "(accumulateZip " ++ show acc1 ++ " " ++ show acc2 ++ ")"
        Expression _ p -> "(accumulateOther " ++ show p ++ ")"
        Splitting p -> "(splitting " ++ show p ++ ")"
        Using index p -> "(using " ++ show index ++ " " ++ show p ++ ")"

instance Show (Index env a) where
    show = show . indexToInt

indexToInt :: Index env a -> Int
indexToInt Zero = 0
indexToInt (Succ n) = indexToInt n + 1
```

## B.5 Module Language.SPL.OpenCL.Compiler

```
{-# LANGUAGE QuasiQuotes, GADTs, TupleSections, TypeFamilies, NamedFieldPuns,
     FlexibleContexts, FlexibleInstances, OverlappingInstances, ScopedTypeVariables
     #-}

module Language.SPL.OpenCL.Compiler where

import Language.SPL.Intermediate hiding (peek, If, Expression)
import qualified Language.SPL.Intermediate as I
```

```
import qualified Language.SPL.Syntax as S
import qualified Language.SPL.Operator as S
import Language.SPL.Common (mapFst, mapSnd)

import Language.C.Quote.OpenCL
import Language.C.Syntax hiding (OpenCL)
import System.Random
import Control.Monad.State
import Control.Monad.Reader
import Control.Monad.Writer
import Text.PrettyPrint.Mainland
import Data.Maybe
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Map (Map)
import qualified Data.Map as Map

import qualified Debug.Trace as Debug

data Code = Code {
    statements :: [Stm],
    pairTypes :: Set (S.SPLType, S.SPLType)
    }

instance Monoid Code where
    mempty = Code [] Set.empty
    mappend (Code statements pairTypes) (Code statements' pairTypes') =
        Code (statements ++ statements') (pairTypes `Set.union` pairTypes')

joinVariables vs1 vs2 = Map.unionWithKey join vs1 vs2
    where
        join variable t1 t2 = if t1 == t2 then t1 else variableError
            ("Conflicting types for variable " ++ variable ++ ": " ++ show t1 ++ " "
                 ++ show t2) t1

variableError :: String -> a -> a
variableError = error
--variableError s = Debug.trace ("****************** " ++ s ++ "
     ******************")

newtype Expression a = Expression Exp
newtype Name a = Name String
type M a = StateT (Int, (Map String S.SPLType, Set String)) (Writer Code) a


data family Named :: * -> *
data instance Named () = NamedEmpty
data instance Named (env, b) = NamedBind (Named env) (Name b)


class Arguments env where
    arguments :: Int -> (Named env, [Param])

instance Arguments () where
    arguments _ = (NamedEmpty, [])

instance forall env a. (Arguments env, S.Type a) => Arguments (env, a) where
    arguments i = (NamedBind (environment) name, [$cparam|$ty:parameterType $id:x|]
         : parameters)
        where
            parameterType = typeOf name
            name = Name x :: Name a
```

```
            x = "argument_" ++ show i
            (environment, parameters) = arguments (i + 1)


newtype OpenCL env = OpenCL String

openCL :: forall a. (Translate () a, Arguments (EnvironmentOf () a), ResultOf a ~
     Double) =>
    S.Time -> a -> OpenCL (EnvironmentOf () a)
openCL timeStep f =
    let intermediate = translate f :: Intermediate (EnvironmentOf () a) Double in
    let (namedEnvironment, parameters) = arguments 0 in
    let code = functionM timeStep parameters (openCL' [Name "generator"]
         namedEnvironment intermediate) in
    OpenCL (
        "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n" ++
        "#include \"random.cl\"\n" ++
        "#include \"sum.cl\"\n" ++
        show (ppr code))

functionM :: S.Time -> [Param] -> M (Expression Double) -> [Definition]
functionM timeStep parameters m =
    let assigned = ["generator", "time_step"] ++ ["argument_" ++ show i | i <- [0 ..
          length parameters - 1]] in
    let (variables, _, statements, pairTypes, Expression c) = runM (Set.fromList
         assigned) m in
    let declarations = map declaration (Map.toList variables) in
    let pairs = map pairStruct (Set.toList pairTypes) in
    [$cunit|
        $edecls:pairs

        kernel void simulate(
            ulong seed,
            local double * local_means,
            local double * local_standard_deviations,
            global double * global_means,
            global double * global_standard_deviations,
            $params:parameters)
        {
            double time_step = $exp:timeStep;
            struct generator_t generator = initialize(seed);
            $decls:declarations
            $stms:statements
            emit_result(
                $exp:c,
                local_means,
                local_standard_deviations,
                global_means,
                global_standard_deviations
            );
        }

    |]

runM :: Set String -> M a -> (Map String S.SPLType, Set String, [Stm], Set (S.
     SPLType, S.SPLType), a)
runM assigned m =
    let w = runStateT m (0, (Map.empty, assigned)) in
    let ((a, (_, (needsDeclaration, assigned))), Code { statements, pairTypes }) =
         runWriter w in
    (needsDeclaration, assigned, statements, pairTypes, a)
```

```
runM' :: M a -> M (Map String S.SPLType, Set String, [Stm], a)
runM' m = do
    s <- get
    let w = runStateT m s
    let ((a, (i, (needsDeclaration, assigned))), Code { statements, pairTypes }) =
         runWriter w
    lift $ tell (Code [] pairTypes)
    modify (mapFst (const i))
    return (needsDeclaration, assigned, statements, a)

expression :: forall a. S.Type a => Exp -> M (Expression a)
expression e = do
    usePair (Name "" :: Name a)
    return $ Expression e

openCL' :: forall env a. S.Type a =>
    [Name StdGen] -> Named env -> Intermediate env a -> M (Expression a)
openCL' generators environment intermediate = case intermediate of
    Uniform -> do
        let g = head generators
        Expression id <- use g
        e <- expression [$cexp|uniform(&$exp:id)|]
        r <- bind "uniform" e :: M (Name Double)
        use r
    Normal -> do
        let g = head generators
        Expression id <- use g
        e <- expression [$cexp|normal(&$exp:id)|]
        r <- bind "normal" e :: M (Name Double)
        use r
    Let e1 e2 -> do
        c1 <- openCL' generators environment e1
        x <- bind "x" c1
        openCL' generators (NamedBind environment x) e2
    Variable index ->
        use (peek index environment)
    Constant (S.Double value) -> expression [$cexp|$exp:value|]
    Constant (S.Bool False) -> expression [$cexp|0|]
    Constant (S.Bool True) -> expression [$cexp|1|]
    Unary op e1 -> do
        c1 <- openCL' generators environment e1
        unary op c1
    Binary op e1 e2 -> do
        c1 <- openCL' generators environment e1
        c2 <- openCL' generators environment e2
        binary op c1 c2
    I.If e1 e2 e3 -> do
        Expression c1 <- openCL' generators environment e1
        (vs2, as2, ss2, e@(Expression c2)) <- runM' $ openCL' generators environment
             e2
        (vs3, as3, ss3, Expression c3) <- runM' $ openCL' generators environment e3
        mapM statement ss2
        mapM statement ss3
        promiseAssigned (as2 `Set.intersection` as3)
        needsDeclaration vs2
        needsDeclaration vs3
        let returnType = typeOf e
        expression [$cexp|($ty:returnType) select(($ty:returnType) $exp:c3, ($ty:
            returnType) $exp:c2, (ulong) $exp:c1)|]
    Prefix usingTime e1 accumulator -> do
        c1 <- openCL' generators environment e1
        delta <- variable "delta" :: M (Name S.Time)
```

```
        end <- variable "end" :: M (Name Double)
        steps <- variable "steps" :: M (Name Double)
        step@(Name step'') <- variable "step" :: M (Name Double)
        time <- variable "time" :: M (Name Double)
        (initialization, before, insides, after, usingDeltaTime, usesAccumulator,
             accumulator) <-
            prefix generators environment time delta accumulator False
        let needsLoop = not (null insides)
        when (usingDeltaTime || usingTime || needsLoop) $ assign end c1
        when usingTime $ do
            end' <- use end
            assign time end'
        when (usingDeltaTime || needsLoop) $ do
            Expression end' <- use end
            assign steps $ Expression [$cexp|ceil($exp:end' / time_step)|]
            Expression steps' <- use steps
            assign delta $ Expression [$cexp|$exp:steps' == 0 ? time_step : ($exp:
                 end' / $exp:steps')|]
        initialization
        before
        when needsLoop $ do
            (vs, as, ss, _) <- runM' (sequence insides)
            updateTime <- if usingTime
                then do
                    Expression delta' <- use delta
                    assign time $ Expression [$cexp|-$exp:delta'|]
                    Expression time' <- use time
                    return [[$cstm|$exp:time' += $exp:delta';|]]
                else return []
            needsDeclaration (Map.singleton step'' S.DoubleType)
            promiseAssigned (Set.singleton step'')
            Expression steps' <- use steps
            Expression step' <- use step
            statement [$cstm|
                for ($exp:step' = 0; $exp:step' <= $exp:steps'; $exp:step' += 1) {
                    $stms:updateTime
                    $stms:ss
                }
            |]
            needsDeclaration vs
            promiseAssigned as
        after
        use accumulator
    Split e -> do
        Expression id <- use (head generators)
        c <- expression [$cexp|split(&$exp:id)|]
        x <- bind "generator" c :: M (Name StdGen)
        openCL' generators (NamedBind environment x) e
    Use index e -> do
        id <- use (peek index environment)
        g <- bind "generator" id
        openCL' (g:generators) environment e


prefix :: [Name StdGen] -> Named env -> Name S.Time -> Name S.Time -> Accumulator
     env a -> Bool -> M (M (), M (), [M ()], M (), Bool, Bool, Name a)
prefix generators environment time delta process dependsOnAccumulator = case process
      of
    Accumulate e1 useDt e2 process -> do
        let dependsOnAccumulator' = dependsOnAccumulator || isJust e2
        (initialization, before, insides, after, usingDt, mustBeInALoop, result) <-
             case process of
```

```
        Just process -> prefix generators environment time delta process
             dependsOnAccumulator'
        Nothing -> return (return (), return (), [], return (), False, False,
            error "Accumulate with no process: this variable should not be used"
                 )
    let mustBeInALoop' = mustBeInALoop || isJust e2
    accumulator <- variable "accumulator"
    let initialization' = when mustBeInALoop initialization
    let initialization'' = case e2 of
            Just e2 -> do
                initialValue <- openCL' generators environment e2
                assign accumulator initialValue
            Nothing -> return ()
    let inside = do
            let environment' = NamedBind (NamedBind (NamedBind environment delta
                 ) accumulator) result
            body <- openCL' generators environment' e1
            assign accumulator body
    let (before', inside', after') = move inside dependsOnAccumulator'
         mustBeInALoop'
    return (initialization' >> initialization'', before >> before', insides ++
         inside', after >> after',
        usingDt || useDt, mustBeInALoop', accumulator)
Zip process1 process2 -> do
    (initialization1, before1, insides1, after1, usingDt1, mustBeInALoop1,
         result1) <-
        prefix generators environment time delta process1 dependsOnAccumulator
    (initialization2, before2, insides2, after2, usingDt2, mustBeInALoop2,
         result2) <-
        prefix generators environment time delta process2 dependsOnAccumulator
    let mustBeInALoop = mustBeInALoop1 || mustBeInALoop2
    x@(Name x') <- variable "zip"
    let inside = do
            e1 <- use result1
            e2 <- use result2
            e <- pairOf e1 e2
            assign x e
    let (before', inside', after') = move inside dependsOnAccumulator
         mustBeInALoop
    return (
        initialization1 >> initialization2,
        before1 >> before2 >> before',
        insides1 ++ insides2 ++ inside',
        after1 >> after2 >> after',
        usingDt1 || usingDt2,
        mustBeInALoop,
        x)
I.Expression usesTime e -> do
    let environment' = NamedBind environment time
    x <- variable "expression"
    let inside = do
            c <- openCL' generators environment' e
            assign x c
    let canNotMoveOutDirectly = usesTime || stochastic e
    let (before', inside', after') = move inside dependsOnAccumulator
         canNotMoveOutDirectly
    return (return (), before', inside', after', False, dependsOnAccumulator &&
         canNotMoveOutDirectly, x)
Using index process -> do
    e' <- use (peek index environment)
    g <- bind "generator" e'
    prefix (g:generators) environment time delta process dependsOnAccumulator
```

```
    Splitting process -> do
        Expression id <- use (head generators)
        let c = Expression [$cexp|split(&$exp:id)|]
        x <- bind "generator" c :: M (Name StdGen)
        prefix generators (NamedBind environment x) time delta process
             dependsOnAccumulator

move :: M () -> Bool -> Bool -> (M (), [M ()], M ())
move inside dependsOnAccumulator mustBeInALoop =
    case (dependsOnAccumulator, mustBeInALoop) of
        (True, True) -> (return (), [inside], return ())
        (True, False) -> (inside, [], return ())
        (False, True) -> (return (), [], inside)
        (False, False) -> (return (), [], inside)


peek :: Index env a -> Named env -> Name a
peek Zero (NamedBind _ a) = a
peek (Succ n) (NamedBind environment _) = peek n environment


statement :: Stm -> M ()
statement statement' = lift (tell (Code [statement'] Set.empty))


variable :: forall a. S.Type a => String -> M (Name a)
variable baseName = do
    n <- liftM fst get
    modify (mapFst (+ 1))
    let x = baseName ++ "_" ++ show n
    return (Name x) :: M (Name a)

assign :: forall a. S.Type a => Name a -> Expression a -> M ()
assign (Name x) (Expression c) = do
    statement $ [$cstm|$id:x = $exp:c;|]
    let variableType = S.splType (error "typeOf: Type case" :: a)
    modify (mapSnd (\(needsDeclaration, assigned) ->
        (Map.singleton x variableType `joinVariables` needsDeclaration, Set.insert x
             assigned)))

bind :: S.Type a => String -> Expression a -> M (Name a)
bind baseName e = do
    name <- variable baseName
    assign name e
    return name

use :: S.Type a => Name a -> M (Expression a)
use (Name x) = do
    existing <- liftM (Set.member x . snd . snd) get
    when (not existing) $ variableError ("Variable " ++ x ++ " has not yet been
         initialized") (return () :: M ())
    expression [$cexp|$id:x|]


needsDeclaration :: Map String S.SPLType -> M ()
needsDeclaration vs = modify $ mapSnd $ mapFst $ joinVariables vs

promiseAssigned :: Set String -> M ()
promiseAssigned as = modify $ mapSnd $ mapSnd $ Set.union as


declaration :: (String, S.SPLType) -> InitGroup
```

```
declaration (x, t) =
    let variableType = typeId t in
    [$cdecl|$ty:variableType $id:x;|]


splTypeOf :: forall b a. S.Type a => b a -> S.SPLType
splTypeOf _ = S.splType (error "typeOf: Type case" :: a)

typeOf :: S.Type a => b a -> Type
typeOf object = typeId (splTypeOf object)

typeId :: S.SPLType -> Type
typeId t = case t of
    S.DoubleType -> [$cty|double|]
    S.BoolType -> [$cty|int /* bool */|]
    S.GeneratorType -> [$cty|struct generator_t|]
    p@(S.PairType _ _) ->
        let x = typeName p in
        [$cty|struct $id:x|]

typeName :: S.SPLType -> String
typeName t = typeName' t ++ "_t"
    where
        typeName' t = case t of
            S.DoubleType -> "double"
            S.BoolType -> "bool"
            S.GeneratorType -> error "No name for generators"
            S.PairType t1 t2 -> "pair_" ++ typeName' t1 ++ "_" ++ typeName' t2

usePair :: forall a. S.Type a => Name a -> M ()
usePair _ = case S.splType (error "usePair" :: a) of
    S.PairType t1 t2 -> lift (tell (Code [] (Set.singleton (t1, t2))))
    _ -> return ()

pairStruct :: (S.SPLType, S.SPLType) -> Definition
pairStruct (t1, t2) = do
    let name = typeName (S.PairType t1 t2)
    let first = typeId t1
    let second = typeId t2
    [$cedecl|
        struct $id:name {
            $ty:first first;
            $ty:second second;
        };
    |]

unary :: forall a b. (S.Type a, S.Type b) =>
    S.UnaryOperator a b -> Expression a -> M (Expression b)
unary op (Expression e) = expression $ case op of
    S.Negate -> [$cexp|-$exp:e|]
    S.Abs -> [$cexp|fabs($exp:e)|]
    S.Sign -> [$cexp|sign($exp:e)|]
    S.Exp -> [$cexp|exp($exp:e)|]
    S.Log -> [$cexp|log($exp:e)|]
    S.Sqrt -> [$cexp|sqrt($exp:e)|]
    S.Sin -> [$cexp|sin($exp:e)|]
    S.Cos -> [$cexp|cos($exp:e)|]
    S.Tan -> [$cexp|tan($exp:e)|]
    S.Asin -> [$cexp|asin($exp:e)|]
    S.Acos -> [$cexp|acos($exp:e)|]
    S.Atan -> [$cexp|atan($exp:e)|]
    S.Sinh -> [$cexp|sinh($exp:e)|]
```

```
    S.Cosh -> [$cexp|cosh($exp:e)|]
    S.Tanh -> [$cexp|tanh($exp:e)|]
    S.Asinh -> [$cexp|asinh($exp:e)|]
    S.Acosh -> [$cexp|acosh($exp:e)|]
    S.Atanh -> [$cexp|atanh($exp:e)|]
    S.Not -> [$cexp|!($exp:e)|]
    S.First -> [$cexp|$exp:e.first|]
    S.Second -> [$cexp|$exp:e.second|]


binary :: forall a b c. (S.Type a, S.Type b, S.Type c) =>
    S.BinaryOperator a b c -> Expression a -> Expression b -> M (Expression c)
binary op e1'@(Expression e1) e2'@(Expression e2) = expression $ case op of
    S.Add -> [$cexp|$exp:e1 + $exp:e2|]
    S.Sub -> [$cexp|$exp:e1 - $exp:e2|]
    S.Mult -> [$cexp|$exp:e1 * $exp:e2|]
    S.Div -> [$cexp|$exp:e1 / $exp:e2|]
    S.Min -> [$cexp|min($exp:e1, $exp:e2)|]
    S.Max -> [$cexp|max($exp:e1, $exp:e2)|]
    S.Power -> [$cexp|pow($exp:e1, $exp:e2)|]
    S.LogBase -> [$cexp|log($exp:e2) / log($exp:e1)|]
    S.Less -> [$cexp|isless($exp:e1, $exp:e2)|]
    S.LessEqual -> [$cexp|islessequal($exp:e1, $exp:e2)|]
    S.Greater -> [$cexp|isgreater($exp:e1, $exp:e2)|]
    S.GreaterEqual -> [$cexp|isgreaterequal($exp:e1, $exp:e2)|]
    S.Equal -> [$cexp|isequal($exp:e1, $exp:e2)|]
    S.NotEqual -> [$cexp|isnotequal($exp:e1, $exp:e2)|]
    S.And -> [$cexp|$exp:e1 & $exp:e2|]
    S.Or -> [$cexp|$exp:e1 | $exp:e2|]
    S.Pair -> pairOf' e1' e2'


pairOf :: forall a b. (S.Type a, S.Type b) =>
    Expression a -> Expression b -> M (Expression (a, b))
pairOf e1 e2 = expression $ pairOf' e1 e2

pairOf' :: forall a b. (S.Type a, S.Type b) =>
    Expression a -> Expression b -> Exp
pairOf' (Expression e1) (Expression e2) =
    let pairId = typeId (S.PairType (S.splType (error "binary" :: a)) (S.splType (
        error "binary" :: b))) in
    [$cexp|($ty:pairId) {$exp:e1, $exp:e2}|]
```

## B.6 Module Language.SPL.OpenCL.Runner

```
{-# LANGUAGE GADTs, FlexibleInstances, FlexibleContexts, MultiParamTypeClasses,
    ScopedTypeVariables, TypeFamilies, EmptyDataDecls #-}
module Language.SPL.OpenCL.Runner (compile, compileSource, Phantom) where

import Language.SPL.Intermediate
import Language.SPL.OpenCL.Compiler
import Language.SPL.SimulationResult
import Language.SPL

import System.OpenCL
import Foreign hiding (mallocArray)
import Foreign.C
import Control.Monad
import Control.Applicative
import Prelude hiding (lookup, zip, reverse)
import qualified Prelude as P
```

```
import Data.Word
import System.Random

printDebug = False

deviceTypes = [DeviceTypeGPU]
properties = [QueueOutOfOrderExec]

defaultGroupSize = 512
defaultGroupCount = 512 * 4


data Phantom a


class Function a where
    type FunctionOf a
    toFunction :: Phantom a -> [KernelArg] -> ([KernelArg] -> IO SimulationResult)
        -> FunctionOf a


instance Function (Dist a) where
    type FunctionOf (Dist a) = IO SimulationResult
    toFunction _ arguments f = f arguments

instance (Function b, Storable a) => Function (Dist a -> b) where
    type FunctionOf (Dist a -> b) = a -> FunctionOf b
    toFunction _ arguments f = \a -> toFunction (undefined :: Phantom b) (VArg a :
        arguments) f


instance (Function b, Storable a) => Function (Process a -> b) where
    type FunctionOf (Process a -> b) = a -> FunctionOf b
    toFunction _ arguments f = \a -> toFunction (undefined :: Phantom b) (VArg a :
        arguments) f


compile :: forall a. (
    Translate () a,
    ResultOf a ~ Double,
    Function a,
    Arguments (EnvironmentOf () a)) =>
    Time -> a -> IO (FunctionOf a)
compile timeStep distribution = do
    let source = openCL timeStep distribution
    compileSource (undefined :: Phantom a) defaultGroupSize defaultGroupCount source

compileSource :: forall a. (Translate () a, Function a) => Phantom a -> Int -> Int
     -> OpenCL (EnvironmentOf () a) -> IO (FunctionOf a)
compileSource _ groupSize groupCount (OpenCL source) = do
    debugPrint source
    platforms <- getPlatforms
    debugPrint (show platforms)
    let platform = head platforms
    devices' <- getDevices platform deviceTypes
    let devices = take 1 (devices')
    (debugPrint . show) =<< mapM deviceName devices
    context <- createContext [ContextPlatform platform] devices
    queues <- mapM (\device -> createCommandQueue context device properties) devices
    program <- createProgram context source
    debugPrint "Program created"
    buildProgram program devices "-Werror -ILanguage/SPL/include"
```

```
debugPrint "Program built"
simulateKernels <- mapM (const (createKernel program "simulate")) queues
debugPrint "Created simulate kernels"
resultKernels <- mapM (const (createKernel program "global_result")) queues
debugPrint "Created global_result kernels"
debugPrint "Kernels created"
let units = P.zip3 queues simulateKernels resultKernels
let groupCount' = groupCount `div` length units -- TODO
let workPerDevice = groupCount' * groupSize
-------------------------
return $ toFunction (undefined :: Phantom a) [] $ \arguments -> do
    allocateArrays (length units) $ \arrays -> do
        events <- mapM (enqueueKernel context groupCount' arguments) (P.zip
            units arrays)

        waitForEvents (concat events)

        let (meansArray, standardDeviationsArray) = unzip arrays
        means <- mapM (\array -> liftM head $ peekArray 1 array :: IO Double)
            meansArray
        standardDeviations <- mapM (\array -> liftM head $ peekArray 1 array ::
            IO Double) standardDeviationsArray
        return $ combine workPerDevice means standardDeviations
where
    doubleSize = sizeOf (error "sizeOf" :: Double)

    allocateArrays = allocateArrays' []
    allocateArrays' arrays 0 function = function (P.reverse arrays)
    allocateArrays' arrays n function =
        allocaArray 1 $ \array1 ->
            allocaArray 1 $ \array2 ->
                allocateArrays' ((array1, array2):arrays) (n - 1) function

    enqueueKernel context groupCount' arguments ((queue, simulateKernel,
         resultKernel), (meansArray, standardDeviationsArray)) = do
        seed <- randomRIO (fromIntegral (minBound :: Word64), fromIntegral (
            maxBound :: Word64)) :: IO Integer
        let seed' = fromIntegral seed :: Word64

        let workPerDevice = groupCount' * groupSize

        meansBuffer <- mallocArray context [MemObjectReadWrite] groupCount' ::
            IO (MemObject Double)
        standardDeviationsBuffer <- mallocArray context [MemObjectReadWrite]
            groupCount' :: IO (MemObject Double)

        setKernelArgs simulateKernel ([
            VArg seed',
            HollowArg (fromIntegral $ groupSize * doubleSize),
            HollowArg (fromIntegral $ groupSize * doubleSize),
            MObjArg meansBuffer,
            MObjArg standardDeviationsBuffer
            ] ++ arguments)
        simulateEvent <- enqueueNDRangeKernel queue simulateKernel []
            [fromIntegral $ workPerDevice] [fromIntegral $ groupSize]
            []

        setKernelArgs resultKernel [
            VArg (fromIntegral $ groupSize :: Int32),
            MObjArg meansBuffer,
            MObjArg standardDeviationsBuffer]
        resultEvent <- enqueueNDRangeKernel queue resultKernel []
```

```
        [fromIntegral $ groupCount' `div` 2] [fromIntegral $ groupCount' `
            div` 2]
        [simulateEvent]

    meansEvent <- enqueueReadBuffer queue meansBuffer False 0 (fromIntegral
        $ 1 * doubleSize) meansArray [resultEvent]
    standardDeviationsEvent <- enqueueReadBuffer queue
        standardDeviationsBuffer False 0 (fromIntegral $ 1 * doubleSize)
        standardDeviationsArray [resultEvent]

    return [meansEvent, standardDeviationsEvent]

debugPrint :: String -> IO ()
debugPrint msg = when printDebug (putStrLn msg)
```

# Appendix C

# Unit test code

## C.1 Module Language.SPL.Test.UnitTests

```
module Language.SPL.Test.UnitTests where

import Language.SPL
import Language.SPL.Test.TestUtilities

import Test.HUnit
import Prelude hiding (lookup, map, zip)

testList = [
    distBasicTests,
    distUnaryTests,
    distBinaryTests,
    processTests,
    accumulatorTests,
    skipTests,
    propertiesTests,
    bugTests
    ]


distBasicTests = TestLabel "Dist constructs" $ TestList [
    testFuzzy "Normal" 0 normal,
    testFuzzy "Uniform" 0.5 uniform,
    testExact "Constant Double" 42 42,
    testExact "if True" 1 (if_ true 1 0),
    testExact "if False" 0 (if_ false 1 0),
    testExact "lookup" 24 (lookup 24 time),
    testExact "sample" 0 (uniform `sample` \d -> d - d),
    testExact "Without sample" 0 (if_ (uniform - uniform .==. 0) 1 0)
    ]


distUnaryTests = TestLabel "Dist unary operators" $ TestList [
    testExact "Negate" (-1) (negate 1),
    testExact "Floating abs" 1 (abs (1 :: Dist Double)),
    testExact "Floating abs" 1 (abs (-(1 :: Dist Double))),
    testExact "Sign" (-1) (signum (-1.2)),
    testExact "Sign" 1 (signum 1.2),
    testExact "if True" 1 (if_ true 1 0),
    testExact "if False" 0 (if_ false 1 0),
```

```
    testExact "Exp" (exp 13.37) (exp 13.37),
    testExact "Log" (log 13.37) (log 13.37),
    testExact "Sqrt" 7 (sqrt 49),
    testExact "Sin" (sin 0.5) (sin 0.5),
    testExact "Cos" (cos 0.5) (cos 0.5),
    testExact "Tan" (tan 0.5) (tan 0.5),
    testExact "Asin" (asin 0.5) (asin 0.5),
    testExact "Acos" (acos 0.5) (acos 0.5),
    testExact "Atan" (atan 0.5) (atan 0.5),
    testExact "Sinh" (sinh 0.5) (sinh 0.5),
    testExact "Cosh" (cosh 0.5) (cosh 0.5),
    testExact "Asinh" (asinh 0.5) (asinh 0.5),
    testExact "Acosh" (acosh 2) (acosh 2)
    ]


distBinaryTests = TestLabel "Dist binary operators" $ TestList [
    testExact "Add" 8 (3 + 5),
    testExact "Sub" 5 (8 - 3),
    testExact "Mult" 120 (3 * 5 * 8),
    testExact "Div" 3 (120 / 8 / 5),
    testExact "Min" 5 (min_ 5 8),
    testExact "Min" 5 (min_ 8 5),
    testExact "Max" 8 (max_ 5 8),
    testExact "Max" 8 (max_ 8 5),
    testExact "Power" 1024 (2 ** 10),
    testExact "LogBase" 2 (logBase 8 64),
    testExact "Less True" 1 (if_ (5 .<. (8 :: Dist Double)) 1 0),
    testExact "Less False" 0 (if_ (5 .<. (5 :: Dist Double)) 1 0),
    testExact "Less False" 0 (if_ (8 .<. (5 :: Dist Double)) 1 0),
    testExact "Less equals True" 1 (if_ (5 .<=. (8 :: Dist Double)) 1 0),
    testExact "Less equals False" 1 (if_ (5 .<=. (5 :: Dist Double)) 1 0),
    testExact "Less equals False" 0 (if_ (8 .<=. (5 :: Dist Double)) 1 0),
    testExact "Greater True" 1 (if_ (8 .>. (5 :: Dist Double)) 1 0),
    testExact "Greater False" 0 (if_ (5 .>. (5 :: Dist Double)) 1 0),
    testExact "Greater False" 0 (if_ (5 .>. (8 :: Dist Double)) 1 0),
    testExact "Greater equals True" 1 (if_ (8 .>=. (5 :: Dist Double)) 1 0),
    testExact "Greater equals False" 1 (if_ (5 .>=. (5 :: Dist Double)) 1 0),
    testExact "Greater equals False" 0 (if_ (5 .>=. (8 :: Dist Double)) 1 0),
    testExact "Equals True" 1 (if_ (5 .==. (5 :: Dist Double)) 1 0),
    testExact "Equals False" 0 (if_ (5 .==. (8 :: Dist Double)) 1 0),
    testExact "Equals False" 0 (if_ (8 .==. (5 :: Dist Double)) 1 0),
    testExact "Not equals True" 1 (if_ (8 ./=. (5 :: Dist Double)) 1 0),
    testExact "Not equals True" 1 (if_ (5 ./=. (8 :: Dist Double)) 1 0),
    testExact "Not equals False" 0 (if_ (5 ./=. (5 :: Dist Double)) 1 0),
    testExact "Not true" 0 (if_ (not_ true) 1 0),
    testExact "Not false" 1 (if_ (not_ false) 1 0),
    testExact "And true true" 1 (if_ (true .&&. true) 1 0),
    testExact "And true false" 0 (if_ (true .&&. false) 1 0),
    testExact "And false true" 0 (if_ (false .&&. true) 1 0),
    testExact "And false false" 0 (if_ (false .&&. false) 1 0),
    testExact "Or true true" 1 (if_ (true .||. true) 1 0),
    testExact "Or true false" 1 (if_ (true .||. false) 1 0),
    testExact "Or false true" 1 (if_ (false .||. true) 1 0),
    testExact "Or false false" 0 (if_ (false .||. false) 1 0),
    testExact "Not true" 0 (if_ (not_ true) 1 0),
    testExact "Not false" 1 (if_ (not_ false) 1 0)
    ]


processTests = TestLabel "Processes constructs" $ TestList [
    testExact "Always" 48 (let p = always 24 in lookup 5 p + lookup 10 p),
```

```
    testExact "Closed natural lookup" 49 (lookup 7 (closed (\t -> t ** 2))),
    testExact "Closed float lookup" 0.25 (lookup 0.5 (closed (\t -> t ** 2))),
    testExact "Closed base case" 0 (lookup 0 time),
    testExact "Closed minimal lookup" 0.0001 (lookup 0.0001 time),
    testExact "First zip" 5 (lookup 10 (first (zip (always 5) (always 8)),
    testExact "Second zip" 8 (lookup 10 (first (zip (always 5) (always 8)),
    testExact "Trace" 1 (lookup 100 (brownian `trace` (\p -> if_ (abs (p - p) .<.
        0.00001) 1 (1/0)))),
    testFuzzy "Without trace" 0 (lookup 100 (if_ (abs (brownian - brownian) .<.
        0.00001) 1 0)),
    -- Unary
    testExact "Negate" (-3) (let p = negate time in lookup 1 p + lookup 2 p),
    testExact "Abs FLoating" 1 (lookup 10 (abs (-1 :: Process Double))),
    -- Binary
    testExact "Add" 18 (lookup 10 (always 3 + always 5 + time)),
    testExact "Less is less" 1 (lookup 4 (if_ (time .<. (always 5)) 1 0)),
    testExact "Less is no more" 0 (lookup 6 (if_ (time .<. (always 5)) 1 0)),
    -- Ternary
    testExact "if True" 1 (lookup 10 (if_ (always true) (always 1) (always 0))),
    testExact "if False" 0 (lookup 10 (if_ (always false) (always 1) (always 0)))
    ]


accumulatorTests = TestLabel "Accumulator optimizations" $ TestList [
    testCheck "Using only delta" $ lookup 1 $ prefix (\d _ _ -> d) 0 (0 :: Process
        Double),
    testExact "Time via trace" 42 $ lookup 42 $ (closed $ \d0 -> d0) `trace` (\p0 ->
         prefix (\_ _ d3 -> d3) 0 p0),
    testApproximate' 1 "Iterative sum" 50 $ lookup 10 $ integral time, -- Slack of 1
         to account for large time steps such as 0.1
    testExact "Map" (-1) $ lookup 10 $ prefix (\_ _ d -> -d) undefined 1,
    testExact "Map Zip" 3 $ lookup 10 $ prefix (\_ _ pair -> first pair + second
        pair) undefined (pair 1 2),
    testExact "Map Zip Time" 20 $ lookup 10 $ time + time,
    testApproximate "Map Accumulate Delta" 10 $ lookup 10 $ iterative (\d a -> a + d
        ) 0,
    testApproximate "Map Zip Accumulate Time" 60 $ lookup 10 $ time + integral time,
    testExact "Zip" 77 $ second $ lookup 10 $ (zip 42 77 :: Process (Double, Double)
        ),
    testApproximate "Accumulate Closed Lookup Closed" (10 * 20) $ lookup 10 $
        integral (closed $ \t -> lookup 20 $ closed id),
    testApproximate "Accumulate Closed Lookup Prefix" (10 * 200) $ lookup 10 $
        integral (closed $ \t -> lookup 20 $ integral time),
    testApproximate "Accumulate Lookup Prefix" (10 * 200) $ lookup 10 $ iterative (\
        d a -> a + d * (lookup 20 $ integral time)) 0,
    testApproximate "Accumulate Normal" 1 $ if_ (lookup 100 brownian .<. 0.2) 1
        (1/0),
    testApproximate "Accumulate Uniform" 50 $ lookup 100 $ iterative (\d a -> a + d
        * uniform) 0,
    testApproximate "Sample Accumulate Uniform" 0 $ uniform `sample` \v -> (-1000 *
        v) + lookup 1000 (integral (always v)),
    testApproximate "Trace Accumulate Brownian" 0 $ lookup 100 $ brownian `trace` \b
         -> integral b - integral b
    ]


skipTests = TestLabel "skip" $ TestList [
    testExact "Skip Constant" 42 $ lookup 5 $ skip 10 42,
    testExact "Skip Time" 15 $ lookup 5 $ skip 10 time,
    testExact "Skip Skip Time" 15 $ lookup 0 $ skip 5 $ skip 10 time,
    testExact "Skip 0" 0 $ lookup 0 $ skip 0 $ inclusivePrefix (\_ a v -> a + 1) 0
        (0 :: Process Double),
```

```
    testApproximate "Skip 1" 1 $ lookup 0 $ skip 1 $ inclusivePrefix (\d a v -> a +
        d) 0 (0 :: Process Double),
    testApproximate "Lookup n (skip m) p == lookup (n + m) p" 200 $ lookup 10 $ skip
         10 $ integral time,
    testApproximate "Accumulate Skip Time" 150 $ lookup 10 $ integral $ skip 10 time
        ,
    testFuzzy "Accumulate Skip Zip Stochastic-Accumulate" 150 $ lookup 10 $ integral
         $ skip 10 (time + brownian * 0.1),
    testApproximate "Skip Accumulate Closed Lookup Closed" 200 $ lookup 10 $ skip 10
         $ integral $ closed (\t -> lookup t time),
    testApproximate "Accumulate Closed Lookup Skip Closed" 150 $ lookup 10 $
        integral $ closed (\t -> lookup t (skip 10 time))
    ]

propertiesTests = TestLabel "Properties" $ TestList [
    --TODO We need delta time in the test below
    let dt = 0.1 in testExact "Prefix at time zero" (5 + dt*10) $ lookup 0 $ prefix
        (\dt a v -> a + dt*v) 5 (time + 10),
    testExact "Inclusive Prefix at time zero" 5 $ lookup 0 $ inclusivePrefix  (\dt a
         v -> dt*v + a) 5 (time + 10),
    testExact "Brownian start at zero" 0 $ lookup 0 brownian,
    testExact "Integral p is zero at time 0" 0 $ lookup 0 (integral (always (1/0))),
    testExact "Integrate is correct on constant processes 1" (0*1000) $ lookup 0 (
        integral (always 1000)),
    testExact "Integrate is correct on constant processes 2" (0.5*1000) $ lookup 0.5
         (integral (always 1000)),
    testExact "Integrate is correct on constant processes 3" (1*1000) $ lookup 1 (
        integral (always 1000)),
    testExact "Integrate is correct on constant processes 4" (10*1000) $ lookup 10 (
        integral (always 1000))
    ]

bugTests = TestLabel "Previous bugs" $ TestList [
    testExact "Traced process used different number of times" 0 $ lookup 100 $
        always uniform `trace` \x ->
            always uniform `trace` \n ->
                let n' = if_ (x .<. 0.5) n 0 in
                integral (assert_ (x .>=. 0.5 .||. n' .==. n)),
    testExact "Equals" 0 $ normal `sample` \n -> lookup 100 $ assert_ (equals (
        always n)),
    testExact "Stochastic calculations are not loop invariant" 0 $ lookup 100 $
        assert_ (not_ (equals (always normal))),
    testExact "If is lazy in the then and else branches" 0 $ lookup 10 $ if_ false
        (1 + always normal) 0,
    testFuzzy "Uniform is sometimes close to 1" 1 $ lookup 1000 $ prefix (\_ a _ ->
        a `max_` uniform) 0 time
    ]

equals :: Process Double -> Process Bool
equals p = map first $ prefix (\_ a v -> pair (second v .==. 0 .||. first a .&&.
     second a .==. first v) (first v)) (pair true 0) (zip p time)

assert_ p = if_ p 0 (1/0)
```

# Appendix D

# Pricing test code

## D.1 Module Language.CC.Test.PricingTest

```
{-# LANGUAGE NoMonomorphismRestriction #-}

import Language.CC.Test.BlackScholes
import Language.CC.Syntax
import Language.CC.Model
import Language.CC
import Language.SPL
import Language.SPL.Interpreter
import qualified Language.SPL.Test.Simulator as S
import qualified Language.SPL.SimulationResult as R
import qualified Language.SPL.OpenCL as O

import Prelude hiding (and, or, truncate, lookup, map)
import qualified Test.QuickCheck as Q
import Test.QuickCheck.Property (morallyDubiousIOProperty)
import Control.Monad
import System.Random
import qualified Data.List as L
import Data.Word
import System (getArgs, getProgName, exitFailure)
import qualified Debug.Trace as Debug


main = do
    args <- getArgs
    case args of
        [] -> usage
        ["alternatives"] -> putStrLn $ L.intercalate "\n" alternatives
        ("all":args) -> testAll =<< getN args
        ("discounting":args) -> testDiscounting =<< getN args
        ("sanity":args) -> testSanity =<< getN args
        ("europeanCall":args) -> testEuropeanCall =<< getN args
        args -> testAll =<< getN args
    where
        alternatives = ["discounting", "sanity", "europeanCall"]
        usage = do
            n <- getProgName
            putStrLn $ "USAGE: " ++ n ++ " [all|" ++ L.intercalate "|" alternatives
                 ++ "] [numberOfTests]"
            exitFailure
```

```
        n = 43
        getN [] = return n
        getN [n] = case (reads n) :: [(Int, String)] of
            [(n, "")] -> return n
            _ -> usage
        getN _ = usage


quickCheckTimes n = Q.quickCheckWith (Q.stdArgs{Q.maxSuccess = n})

testDiscounting n = quickCheckTimes n (Q.label "Discounting" discountingProperty)
testSanity n = quickCheckTimes n (Q.label "Sanity" sanityProperty)
testEuropeanCall n = quickCheckTimes n (Q.label "European call" europeanCallProperty
     )

testAll n = do
    testDiscounting n
    testSanity n
    testEuropeanCall n


---------------------- Properties ----------------------

discountingProperty :: Rate -> Years -> Price -> Q.Property
discountingProperty (Rate rate) (Years t) (Price amount) =
    let contract = get (truncate t (scale (constant amount) (one DKK))) in
    let process = priceProcess (simpleModel (const (constant rate))) DKK contract in
    let referencePrice = amount * exp (-rate * t) in
    testValueNow referencePrice process



-- This test was suggested by Mogens Steffensen, he wrote:
--
-- at 5 underlying
-- priser paa denne skal vaere S eftersom vaerdien af den underliggende om 5 aar
-- skal vaere vaerdien af den underliggende idag. Det er en god test for om I
-- faar modelleret den underliggende rigtigt. Samtidig er det et tjek
-- af kvaliteten af jeres simleringsprocedure.

{-
sanityProperty :: Price -> Years -> Rate -> Volatility -> Q.Property
sanityProperty (Price currentPrice) (Years endTime) (Rate rate) (Volatility
     volatility) =
    let process =
            withIntel (constant currentPrice) (constant rate) (constant volatility)
                 $ \intel ->
            let contract = get (truncate endTime (intel)) in
            priceProcess (simpleModel (const (constant rate))) USD contract in
    testValueNow currentPrice process
-}

sanityProperty :: Price -> Years -> Rate -> Volatility -> Q.Property
sanityProperty (Price currentPrice) (Years endTime) (Rate rate) (Volatility
     volatility) =
    let share = underlying currentPrice rate volatility in
    let contract = get (truncate endTime (scale share)) in
    let process = priceProcess (simpleModel (const (constant rate))) USD contract in
    testValueNow currentPrice process


europeanCallProperty :: Price -> Price -> Years -> Rate -> Volatility -> Q.Property
```

```
europeanCallProperty (Price currentPrice) (Price strikePrice) (Years endTime) (Rate
     rate) (Volatility volatility) = do
    let process =
            withIntel (constant currentPrice) (constant rate) (constant volatility)
                 $ \intel ->
            let usd = constant strikePrice `as` USD in
            let contract = get (truncate endTime ((intel `and` give usd) `or` zero))
                  in
            priceProcess (simpleModel (const (constant rate))) USD contract

    let referencePrice = blackScholesCall currentPrice strikePrice endTime rate
         volatility
    testValueNow referencePrice process


-- The following two processes was suggested by Morgens Steffensen as good
     underlyings.
-- In particular thay should have the property that when we model a european call/
     put we
-- should find the same price at the closed BS formular.

--  S*e^((r-1/2*v^2)*t+v*normal*sqrt(t))
--  Consider: Does it make sense that it isn't the same "normal" on different times?
      Maybe use trace?
withIntel initialPrice rate volatility = traceIn USD $
    initialPrice * exp ((rate - 0.5 * volatility ^ 2) * time + volatility * always
         normal * sqrt(time))

--  S*e^((r-1/2*v^2)*t+v*W(t))                    <----- thank you Mogens!
withIntel' initialPrice rate volatility = traceIn USD $
    always initialPrice * exp (always (rate - 0.5 * volatility ^ 2) * time + always
         volatility * brownian)

underlying s r v = always s * exp (always (r - 0.5 * v ^ 2) * time + always v *
     brownian)

---------------------- Auxiliaries ----------------------

testValueNow :: Double -> Process Double -> Q.Property
testValueNow referencePrice contractProcess =
    let referenceResult = R.SimulationResult referencePrice 0 in
    morallyDubiousIOProperty $ do
        r <- S.requireExpectation R.standardEquality referenceResult (lookup 0
             contractProcess)
        case r of
            Left (error, _) -> do
                putStrLn error
                return False
            Right r -> do
                print r
                return True

instance Q.Arbitrary StdGen where
    arbitrary = liftM mkStdGen Q.arbitrary


data Years = Years Double deriving (Show)

instance Q.Arbitrary Years where
    arbitrary = liftM Years $ Q.choose (0.0, 20.0)

data Price = Price Double deriving (Show)
```

```
instance Q.Arbitrary Price where
    arbitrary = liftM Price (Q.arbitrary `Q.suchThat` (> 0))


data Rate = Rate Double deriving (Show)

instance Q.Arbitrary Rate where
    arbitrary = liftM Rate $ Q.choose (0.00, 0.20)


data Volatility = Volatility Double deriving (Show)

instance Q.Arbitrary Volatility where
    arbitrary = liftM Volatility $ Q.choose (0.00, 0.50)
```

## D.2 Module Language.SPL.Test.AsianTest

```
module Language.SPL.Test.AsianTest where

import Language.SPL hiding (average)
import Language.SPL.Test.TestUtilities

import Test.HUnit
import Prelude hiding (lookup, map)

testList = [asianTests]

-- TODO: This test requires a time step of 0.1 (since that is what is used to
     produce the reference values)

asianOption :: Dist Double -> Dist Double -> Dist Double -> Dist Double -> Dist Time
      -> Dist Double
asianOption initialPrice rate volatility exercisePrice t =
    let d = asianProcess (always initialPrice) (always rate) (always volatility) (
        always exercisePrice) in
    lookup t d

asianProcess :: Process Double -> Process Double -> Process Double -> Process Double
      -> Process Time
asianProcess initialPrice rate volatility exercisePrice =
    let underlying = intel initialPrice rate volatility in
    let option = underlying `trace` (\u -> (average u - exercisePrice) `max_` 0) in
    exp (-rate * time) * option

-- The reason why this average is a bit odd is that it's exclusive of time zero,
     just as in the paper.
average :: Process Double -> Process Double
average process =
    let process' = if_ (time .==. 0) 0 process in
    let sumCount = prefix (\_ acc value -> pair (first acc + value) (second acc + 1)
        ) (pair 0 0) process' in
    map (\p -> first p / (second p - 1)) sumCount

intel initialPrice rate volatility =
    initialPrice * exp ((rate - 0.5 * volatility ^ 2) * time + volatility * brownian
        )


-- The reference values are from Table 1 of
--    "The Pricing of Discretely Sampled Asian and Lookback Options:
```

```
--     A Change of Numeraire Approach"
-- By Jesper Andreasen

asian strikePrice = asianOption 100 0.05 0.20 strikePrice 1

referenceValues = [
    ( 90.0, 12.98),
    ( 92.5, 11.05),
    ( 95.0,  9.27),
    ( 97.5,  7.67),
    (100.0,  6.24),
    (102.5,  5.01),
    (105.0,  3.96),
    (107.5,  3.08),
    (110.0,  2.36)
    ]

asianTests = TestLabel "Asian options (fixed strike)" $ TestList [
    testApproximate ("Strike price " ++ show strikePrice) reference (asian
        strikePrice) |
    (strikePrice, reference) <- referenceValues
    ]
```

## D.3 Module Language.SPL.Test.LookbackTest

```
module Language.SPL.Test.LookbackTest where

import Language.SPL
import Language.SPL.Test.TestUtilities

import Test.HUnit
import Prelude hiding (lookup, map)

testList = [fixedTests, floatingTests]

-- TODO: This test requires a time step of 0.1 (since that is what is used to
     produce the reference values)

fixedProcess :: Process Double -> Process Double -> Process Double -> Process Double
      -> Process Time
fixedProcess initialPrice rate volatility exercisePrice =
    let underlying = intel initialPrice rate volatility in
    let option = underlying `trace` (\u -> (maxValue u - exercisePrice) `max_` 0) in
    exp (-rate * time) * option

fixedOption :: Dist Double -> Dist Double -> Dist Double -> Dist Double -> Dist Time
      -> Dist Double
fixedOption initialPrice rate volatility exercisePrice t =
    let d = fixedProcess (always initialPrice) (always rate) (always volatility) (
        always exercisePrice) in
    lookup t d


floatingProcess :: Process Double -> Process Double -> Process Double -> Process
     Double -> Dist Double -> Process Time
floatingProcess initialPrice rate volatility alpha t =
    let underlying = intel initialPrice rate volatility in
    let option = underlying `trace` (\u -> (maxValue u - alpha * always (lookup t u)
        ) `max_` 0) in
    exp (-rate * time) * option
```

```
floatingOption :: Dist Double -> Dist Double -> Dist Double -> Dist Double -> Dist
     Double -> Dist Double
floatingOption initialPrice rate volatility alpha t =
    let d = floatingProcess (always initialPrice) (always rate) (always volatility)
         (always alpha) t in
    lookup t d


-- The reason why this max is a bit odd is that it's exclusive of time zero, just as
      in the paper.
maxValue :: Process Double -> Process Double
maxValue process =
    let process' = if_ (time .==. 0) (-1/0) process in
    prefix (\_ acc value -> max_ acc value) (-1/0) process'


intel initialPrice rate volatility =
    initialPrice * exp ((rate - 0.5 * volatility ^ 2) * time + volatility * brownian
        )


-- The reference values are from Table 3 and 4 of
--    "The Pricing of Discretely Sampled Asian and Lookback Options:
--     A Change of Numeraire Approach"
-- By Jesper Andreasen

fixed strikePrice = fixedOption 100 0.05 0.20 strikePrice 1
floating strikePrice = floatingOption 100 0.05 0.20 strikePrice 1

fixedReferenceValues = [
    ( 90.0, 24.41),
    ( 92.5, 22.07),
    ( 95.0, 19.78),
    ( 97.5, 17.57),
    (100.0, 15.48),
    (102.5, 13.53),
    (105.0, 11.75),
    (107.5, 10.14),
    (110.0,  8.71)
    ]

floatingReferenceValues = [
    (1.000, 10.01),
    (1.025,  8.27),
    (1.050,  6.77),
    (1.075,  5.51),
    (1.100,  4.46),
    (1.125,  3.59),
    (1.150,  2.88),
    (1.175,  2.30),
    (1.200,  1.83)
    ]

fixedTests = TestLabel "Tests of lookback options with fixed strike" $ TestList $ [
    testApproximate ("Price " ++ show strikePrice) reference (fixed strikePrice) |
    (strikePrice, reference) <- fixedReferenceValues
    ]

floatingTests = TestLabel "Tests of lookback options with floating strike" $
     TestList $ [
    testApproximate ("Alpha " ++ show alpha) reference (floating alpha) |
    (alpha, reference) <- floatingReferenceValues
```

```
    ]
```

## D.4 Module Language.SPL.Test.BasketTest

```
module Language.SPL.Test.BasketTest where

import Language.SPL
import Language.SPL.Test.TestUtilities

import Test.HUnit
import Prelude hiding (lookup, map)

testList = [basketTests]

basketTests = TestLabel "Basket option" $ TestList [
    testApproximate ("Basket test") 0.31 (basket t)
    ]

-- TODO: Requires a time step of 1/200 or so (to match the reference).

-- Ported from: http://stotastic.com/wordpress/2010/05/basket-option-pricing/ (See
     also John Hull chapter 15)
basket t = lookup t $
    pair (always normal) (always normal) `trace` \normals ->
    rate `trace` \rate ->
    let p1 = underlying s1 volatility1 first (pair rate normals) in
    let p2 = underlying s2 volatility2 correlated (pair rate normals) in
    if_ (p1 .>. always k1 .&&. p2 .>. always k2) (exp (-integral rate)) 0

correlated zs = rho * first zs + sqrt(1 - rho ^ 2) * second zs

rate = iterative (\dt r -> r + k * (theta - r) * dt + beta * sqrt dt * normal) r0

underlying initialPrice volatility f pairs =
    inclusivePrefix (\dt s v -> s + s * (first v * dt + volatility * sqrt dt * f (
        second v))) initialPrice pairs

-- Maturity time
t = 1

-- Interest rate model values
k = 0.4
theta = 0.05
beta = 0.03

-- Interest rate initially
r0 = 0.01

-- Strike prices
k1 = 500
k2 = 240

-- Initial prices
s1 = k1
s2 = k2

-- Volatility
volatility1 = 0.2262006
volatility2 = 0.2756421

-- Correlation
```

```
rho = 0.5413732
```