Probabilistic Programming and Discrete Time Markov Chains

Author: Maciej Olejnik

Supervisor: Herbert Wiklicky

Second Marker: Mahdi Cheraghchi

June 14, 2016
Abstract

Probabilistic programming is a steadily developing paradigm of computer science which helps us manage uncertain information. Most random processes can be modelled by a probabilistic program, which can then be supplied to an inference engine to help us predict its outcome. The models are rich and concise, decoupled from their solver, and their development does not require high level of expertise. As a result, the area has found many applications in machine learning.

We present pwc – a compiler for probabilistic programs which stems from our investigation of Linear Operator Semantics. It reduces its input to a Discrete Time Markov Chain, which in turn can be represented by a matrix. One can then analyse the program by multiplying the corresponding arrays of numbers.

The programs will be specified in a probabilistic programming language called pWhile, which we extend with new syntactic constructs as part of this project. We outline how pwc can be used to verify the correctness of a security protocol, as well as help us make decisions and solve probabilistic riddles. Finally, we show that our tool outperforms its predecessor thanks to its back end implemented in a novel, high-performance programming language called Julia.
Acknowledgements

I would like to express my gratitude to my supervisor, Dr Herbert Wiklicky, for his guidance and support throughout this project. I would also like to thank my parents – without them I would not be where I am.
# Contents

1 Introduction 4  
1.1 Motivation ........................................ 4  
1.2 Objectives ........................................ 6  
1.3 Contributions .................................... 6  

2 Background 8  
2.1 Probabilistic programming .......................... 8  
2.2 Mathematical preliminaries ......................... 9  
  2.2.1 Basic probability ................................ 9  
  2.2.2 Markov Chains ................................ 10  
  2.2.3 Kronecker Product .............................. 12  
2.3 Probabilistic While .................................. 13  
2.3.1 Example programs ............................... 14  
2.4 Linear Operator Semantics ........................... 16  
  2.4.1 Probabilistic state ............................. 16  
  2.4.2 Labels ....................................... 18  
  2.4.3 Control flow .................................. 20  
  2.4.4 Local operators ............................... 22  
  2.4.5 Global operators and DTMC generator ........ 28  

3 Design & Language Extension 32  
3.1 Design ............................................. 32  
3.2 Language extensions ................................ 33  
  3.2.1 Arrays ........................................ 33  
  3.2.2 For loop ...................................... 34  
  3.2.3 Case statement ................................ 36  
  3.2.4 Goto statement ................................ 38  
  3.2.5 Constants ..................................... 40  
  3.2.6 Further extensions ............................. 40  
3.3 Language choice .................................... 40  
  3.3.1 pWhile ........................................ 40  
  3.3.2 OCaml ......................................... 42  
  3.3.3 Julia .......................................... 42
Chapter 1

Introduction

This project presents pwc – an OCaml compiler for probabilistic programs, which are written in a probabilistic programming language called pWhile, presented in Section 2.3. The tool stems from Linear Operator Semantics (LOS) – thoroughly described in Section 2.4 – which defines the semantics of pWhile programs in terms of matrix operators. It relies on the finiteness property, and hence every program we will analyse is going to have finitely many variables, each of them having a finite range. Some basic mathematical concepts from areas of probability theory and linear algebra are necessary to fully appreciate LOS – we introduce them in Section 2.2. Finally, probabilistic programming is explored in some detail in Section 2.1.

1.1 Motivation

We motivate our work by presenting some interesting probabilistic problems that can be solved by writing a pWhile program, compiling it with pwc and analysing the output.

We begin with a well-known problem in cryptography which studies secure multi-party communication.

Example 1.1.1 (Dining Cryptographers [3]). Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d’hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other’s right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:

Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see—the one he flipped and the one his left-hand neighbor flipped—fell on the same side or on
different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is.

One can perform a simple case analysis to convince themselves that the protocol is unconditionally secure, i.e. if one of the cryptographers paid for the meal, his identity will not be revealed. Alternatively, one can write a pWhile program modelling the described situation and use pwc to conclude security of the protocol.

The next example is a problem in probability theory with an interesting background.

Example 1.1.2 (Monty Hall [4]). The origins of this example are legendary. Allegedly, it goes back to some TV show in which the contestant was given the chance to win a car or other prizes by picking the right door behind which the desired prize could be found.

The game proceeds as follows: First the contestant is invited to pick one of three doors (behind one is the prize) but the door is not yet opened. Instead, the host – legendary Monty Hall – opens one of the other doors which is empty. After that the contestant is given a last chance to stick with his/her door or to switch to the other closed one. Note that the host (knowing where the prize is) has always at least one door he can open.

The problem is whether it is better to stay stubborn or to switch the chosen door. Assuming that there is an equal chance for all doors to hide the prize it is a favourite exercise in basic probability theory to demonstrate that it is better to switch to a new door.

The same conclusion can also be reached in an alternative way. One can write a pWhile program expressing the probabilistic model corresponding to the presented situation, compile it with pwc and analyse the produced matrices.

Finally, we present several little probabilistic riddles.

Example 1.1.3 (A fair game of chance). The following game is proposed: Alice is given $a$ nickels and Bob is given $b$ nickels initially. At each turn, a dice is thrown - if it comes up heads, Alice gives Bob one nickel, if it comes up tails, Bob gives a nickel to Alice. It goes on until one of them has lost all their money. Now the question is: What are the chances of Alice winning the game?

Example 1.1.4 (Two dice). (1) Throw two dice. If the absolute difference of the two outcomes is 0, 1 or 2, I pay you a nickel, if it’s 3, 4 or 5, you pay me a nickel. Do you wish to play the game?

\footnote{Taken from http://www.cut-the-knot.org/probability.shtml, last retrieved 2016/05/31}
(2) Throw two dice. You win a nickel if 2 or 5 shows on either dice, otherwise you lose a nickel. Do you want to play the game? Bonus: What happens for the n-sided dice?

**Example 1.1.5** (Population expansion). Imagine a population of species in which every member lives exactly one year, however upon death, he gives birth to another two members with probability $\frac{2}{3}$. What is the chance that the family tree of the population, starting with one member, will go on forever?

The pWhile programs modelling the above examples can be found in Section 2.3.1. Another motivation was provided by the recent rise of probabilistic programming as a field of computer science and its many applications, including machine learning and security. We expand on that in Section 2.1.

## 1.2 Objectives

The starting point of the project was the compiler developed by one of the pioneers of Linear Operator Semantics, and the supervisor of this project, Dr Herbert Wiklicky. We often refer to it as an “existing tool” or “reference tool” throughout this report. Our goal was to build similar software, but with improved performance, documentation, tests and following better coding standards. We planned to reuse some parts of the existing tool, however we did not know to what extent. In fact, we intended to start the project by familiarising ourselves with the code base and determining what needs to be redeveloped.

Upon completion of the primary objective, which was producing a sound piece of software, we intended to direct our efforts towards enhancing the performance of the tool. That includes not only refining the code to make it faster, but also using specialised hardware, such as a GPU, to speed up computation. Efficiency was the major deficiency of the reference tool and therefore we intended to put extra effort to improve that aspect of it.

## 1.3 Contributions

During this project, we

- extended the pWhile language with new syntactic constructs (see Section 3.2)
- refactored pwc’s existing front end, thereby improving its functionality and foolproofness (see Sections 4.2 and 4.3)
- implemented from scratch pwc’s back end (see Section 4.4)
- developed an automated test suite specifically for testing pWhile programs (see Section 5)
• created a framework for testing the performance of pwc and comparing the performance of multiple pwc’s implementations (see Section 6.2)
Chapter 2

Background

2.1 Probabilistic programming

Probabilistic programming (PP) is a way to create systems that help us make decisions in the face of uncertainty [11]. Probabilistic programming languages (PPLs) give us means to create those systems, which we will call probabilistic models, and deduce something from them. Majority of PPLs are extensions of classical, deterministic programming languages, such as C, Java or LISP. What they bring in is essentially two things: (i) additional syntax for specifying probability distributions, and (ii) runtime environment which performs probabilistic inference (i.e. reasoning under uncertainty) on the created models. As a result, rather than being executed to produce output, probabilistic programs are supplied to inference engines, which in turn use complex algorithms to draw conclusions from them. One example of a probabilistic programming language is aforementioned pWhile, which extends the well-known While language.

Probabilistic programming originated in 1980s with the hugely theoretical work of D. Kozen [6], C. Jones, G. D. Plotkin [5] or E. Moggi. However, it was only at the start of twenty-first century when first PPLs emerged, with IBAL [10] and Prism [8] leading the way. Nonetheless, they did not receive much attention from the industry and the area was investigated mostly by academics, without many practical uses. That changed in 2013, when PP came into the spotlight as a result of the U.S. Defense Advanced Research Projects Agency (DARPA) launching a four-year program to fund probabilistic programming research. Such a revival is mainly due to the progress in the area of probabilistic inference leading to new tools being developed, as well as the growing complexity of probabilistic models and the need for a new environment to express them. The DARPA program led to many new PPLs, one of them being Picture [7], designed at MIT to easily solve many tasks in computer vision. It has already been shown to outperform many conventional systems when constructing a 3-D model of a human face from 2-D images. Not only is it more efficient at performing this task, but also requires far less code to be written.

Artificial intelligence and specifically machine learning is probably the most
important practical application of probabilistic programming. It is hoped that PP will reduce the complexity of ML applications by introducing a language which enables developers to easily express the theoretical models. That approach could offer similar benefits to machine learning community as introduction of high-level programming languages did to all programmers fifty years ago. It could reduce the time and effort spent on developing code, which would be shorter, faster and more understandable. Moreover, one would no longer need to be an expert in the field to build a successful application. That would lead to more machine learning systems and promote progress in this area. This application of PP is being actively researched at the moment, but if successful, might revolutionise the field.

Another big use of PP is in computer security - many algorithms in public key cryptography use probabilistic encryption and their security depends on the quality of (pseudo)random generator. PP may be used to reason about those algorithms and discover possible deficiencies.

2.2 Mathematical preliminaries

Before we start describing the proposed semantics and its implementation, we briefly cover the underlying mathematical theory. We attempt to motivate each notion we introduce by putting it in the context of probabilistic programming, or more specifically, in the context of the semantics that underlies the implementation of our tool.

2.2.1 Basic probability

One of the most fundamental probabilistic concepts that we must introduce is probability distribution of a random variable. A random variable is a variable whose value is subject to variations due to chance. It can take on a set of possible different values, each with an associated probability. When this set is finite\(^1\), we call such a variable a discrete random variable. One of the main features of programs we consider is that variables are in fact random variables – rather than taking one, definite value at any point of execution, they might take several different values with certain probabilities. As we shall see later, every variable in our programs will have a finite range - we can therefore restrict our attention to discrete random variables. We can then define probability distribution of such a random variable \(X\) as a function assigning probability to each of its possible values. In the discrete case, assuming some natural ordering of the range of \(X\) (which exists when \(X\) takes numeric values) we can represent probability distribution as a vector, which is illustrated by the following example.

Example 2.2.1. Let \(X\) be a random variable with range \(\{1,2,3,4\}\). Suppose \(X\) takes value 1 with probability \(\frac{1}{2}\) and values 3, 4, each with probability \(\frac{1}{4}\). Then

\(^1\)Strictly speaking, we should say finite or countably infinite.
the vector representing the probability distribution of $X$ is

$$v = \left(\frac{1}{2}, 0, \frac{1}{4}, \frac{1}{4}\right)$$

We see that the length of $v$ is equal to the number of possible values of $X$ (which we assumed to be finite), and its $i^{th}$ entry is the probability of $X$ taking $i^{th}$ value in its range. We note that sum of entries of $v$ is equal to 1 – in general, any vector satisfying this property is called a distribution.

One of the main features of probabilistic programming is that variables defined in a program are in fact random variables, i.e. rather than having a definite value at any point of execution, they might take several values with different probabilities. We may therefore abuse the notation slightly and speak of a distribution of a program variable.

### 2.2.2 Markov Chains

Before we shed some light on how to represent the state of the program, we present two notions inherent to our semantics. First of these is Markov chain, i.e. a simple model for a random process that undergoes transitions between different states. For our purposes, Markov chain is assumed to be discrete-time, discrete state-space. In other words, transitions will happen at discrete times and the set of states will be finite. In this setting, we can view a Markov chain as a system which is in a certain state at each step (steps represent moments in time and are often denoted using consecutive natural numbers), with the state changing between steps according to the specified probabilities. One way to represent a discrete-time Markov chain (DTMC) is a diagram and a simple example is depicted in Figure 2.1. It has three states and five transitions.

![Figure 2.1: Simple Markov chain](image)

An important property of Markov chains is that probabilities of transitions from a given state must sum to 1, which can be easily checked to hold for the example from Figure 2.1. Another feature of Markov chains is memorylessness – the next state depends only on the current state. It makes it possible to predict
how the process might behave, and to compute the probabilities of different states occurring at different times.

While a diagram is certainly the most human-readable way to represent a Markov chain, one can also use a matrix to fully describe it. Such a matrix is called a transition matrix. We will also refer to it as a DTMC generator.

**Example 2.2.2** (Generator matrix). Generator of the above Markov chain is

\[
G = \begin{pmatrix}
0 & 1 & 0 \\
0 & \frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & 0 & \frac{1}{2}
\end{pmatrix}
\]

For instance, second row of \(G\) reflects the fact that from state 2, we either move to state 3 or stay at 2, both with equal probabilities (\(\frac{1}{2}\)). In general, \(ij\) – entry of the transition matrix is the probability of moving from state \(i\) to state \(j\).

We note that the generator of a Markov chain is always a stochastic matrix – sum of entries in every row equals 1 (in other words, every row is a distribution). Clearly, from the point of view of implementation, the matrix representation of a Markov chain is preferable. In fact, it constitutes the basis for the whole semantics. The following example introduces the notion of a state vector and shows how generator matrix is used in practice.

**Example 2.2.2** (continued). Imagine a random process \(P\) which can be modelled by a Markov chain from Figure 2.1. It has three states and the probabilities of moving between them are as depicted in the diagram. Suppose that the initial state of \(P\) is 1. We might represent the current state of \(P\) using a (discrete) random variable \(S\). Since the state space is finite, probability distribution of \(S\) may be represented by a vector, which we call a state vector. Let \(v_n\) be the state vector after \(n\) steps of the process. Then

\[
v_0 = (1, 0, 0)
\]

is the initial state vector, since by assumptions the process starts in a first state. In order to compute the state vector after first step of \(P\), one simply post-multiples \(v_0\) by the transition matrix \(G\). Hence

\[
v_1 = v_0 \cdot G = (0, 1, 0)
\]

which means that \(P\) moved to state 2, as expected. Something slightly more interesting happens after second step, when the state vector becomes

\[
v_2 = v_1 \cdot G = (0, \frac{1}{2}, \frac{1}{2})
\]

i.e. the process is either at state 2 or state 3, with equal probabilities.
We will analyse our programs in a very similar manner. In fact, we will view each program as a Markov chain, where different variable values and positions in the program execution give rise to different states, and each executable statement contributes to the generator of the DTMC.

2.2.3 Kronecker Product

The second mathematical notion which emerges repeatedly in our considerations is a binary matrix operator called Kronecker product and denoted by $\otimes$. Given an $n \times m$ matrix $A$ and a $k \times l$ matrix $B$, $A \otimes B$ is the $nk \times ml$ matrix

$$A \otimes B = \begin{pmatrix} a_{1,1} & \ldots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \ldots & a_{n,m} \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & \ldots & b_{1,l} \\ \vdots & \ddots & \vdots \\ b_{k,1} & \ldots & b_{k,l} \end{pmatrix} = \begin{pmatrix} a_{1,1}B & \ldots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \ldots & a_{n,m}B \end{pmatrix}$$

**Example 2.2.3** (Kronecker product). Let

$$A = \begin{pmatrix} 1 & -1 \\ 0 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & -1 \\ -2 & 0 \\ 0 & 2 \end{pmatrix}$$

Then

$$A \otimes B = \begin{pmatrix} 3 & -1 & -3 & 1 \\ -2 & 0 & 2 & 0 \\ 0 & 2 & 0 & -2 \\ -3 & 0 & -4 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

As suggested by the dashed lines, we may view $A \otimes B$ as four copies of $B$, each multiplied by one of the entries of $A$.

The Kronecker product can easily be generalised to an $n$-ary version which is associative but not commutative. For mathematically inclined, we might also observe that Kronecker product is bilinear.

From our point of view, an important property (or the lack of it) is commutativity. In general

$$A \otimes B \neq B \otimes A$$

However, we can turn inequality into equality by introducing a concept of commutation matrix: if $A$ is $n \times m$ and $B$ is $k \times l$, then

$$K_{kn}(A \otimes B)K_{ml} = B \otimes A$$

where $K_{pq}$ is an $pq \times pq$ commutation matrix, defined as

$$K_{pq} = \sum_{i=1}^{p} \sum_{j=1}^{q} (E_{ij} \otimes E_{ij}^T)$$

12
where $E_{ij}$ is a $p \times q$ matrix with $ij$-entry equal to 1 and zeros otherwise, and $E_{ij}^T$ denotes the transpose of $E_{ij}$, i.e. a matrix whose $ij$ – entry is equal to the $ji$ – entry of $E_{ij}$.

We also introduce the mixed-product property of Kronecker product, which states that if $A$, $B$, $C$, $D$ have sizes such that one can form matrix products $AC$ and $BD$, then

$$(A \otimes B)(C \otimes D) = AC \otimes BD$$

### 2.3 Probabilistic While

We now present the syntax of a simple imperative language called pWhile, which extends the well known While [9] with constructs for probabilistic choice and random assignment. It is important for us, because programs written in it serve as input for pwc.

A pWhile program is made up from a possibly empty declaration part $D$ of variables and a single statement $S$ which represents the actual program [4]:

$$P ::= \text{begin } S \text{ end}$$

$$\quad | \quad \text{var } D \text{ begin } S \text{ end}$$

Each declaration in $D$ consists of a variable name $v$ followed by its range $r$, which is either a set of integers or a set $\mathbb{B} = \{\text{true, false}\}$, specified using keyword bool. Formally [4]:

$$r ::= \text{bool}$$

$$\quad | \quad \{c_1, \ldots, c_n\}$$

$$\quad | \quad \{l \ldots h\}$$

$$D ::= v : r$$

$$\quad | \quad v : r ; D$$

We have two different ways of defining the range of a variable. The first of them simply lists the integers making up the range, while the second specifies two end points, separated by dots. The meaning of $\{l \ldots h\}$ is a set of integers $\{l, l+1, \ldots, h\}$ when $l = h$, and an empty set otherwise. The syntax of statements $S$ is as follows [4]:

$$S ::= \text{stop}$$

$$\quad | \quad \text{skip}$$

$$\quad | \quad v := a$$

$$\quad | \quad v \?= r$$

$$\quad | \quad S_1 ; S_2$$

$$\quad | \quad \text{choose } p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}$$

$$\quad | \quad \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

$$\quad | \quad \text{while } b \text{ do } S \text{ od}$$
Most of the above statements should be familiar to the reader. It might be unclear why we introduce two “empty” statements, namely stop and skip. The difference between them is that while skip simply transfers control to the next statement (or terminates the program if there is no statement after it), stop acts as an infinite loop (i.e. transfers control to itself). In Section 3.3.1 we discuss the motivation behind it. We also have two probabilistic statements in pWhile: random assignment $v \equiv r$, which assigns to $v$ a random value from range $r$, with uniform distribution; and probabilistic choice choose, which executes statements $S_1$ and $S_2$ with (normalised) probabilities $\frac{p_1}{p_1+p_2}$ and $\frac{p_2}{p_1+p_2}$, respectively. Probabilities can be given as integers or using the following syntax

$$p ::= \frac{r}{s}$$

to denote probability $p = \frac{r}{s}$, with $r \in \mathbb{Z}$, $s \in \mathbb{Z} \setminus \{0\}$.

We have two types of expressions in pWhile: arithmetic and boolean expressions. The former are of the form

$$a ::= n \quad | \quad a_1 + a_2 \quad | \quad a_1 - a_2 \quad | \quad a_1 \times a_2 \quad | \quad a_1 / a_2 \quad | \quad a_1 \% a_2$$

with $n \in \mathbb{Z}$.

The syntax of boolean expressions is defined by

$$b ::= \text{true} \quad | \quad \text{false} \quad | \quad \text{not} \ b \quad | \quad b_1 \& \& b_2 \quad | \quad b_1 \mid \mid b_2 \quad | \quad a_1 \triangleleft a_2$$

The symbol ‘\$\triangleleft\$’ denotes one of the standard comparison operators for arithmetic expressions, i.e. ‘\$\text{>}\$', ‘\$\text{>=}\$', ‘\$\text{==}\$', ‘\$\text{!=}\$', ‘\$\text{<}\$’ or ‘\$\text{<}\$’.

### 2.3.1 Example programs

To get a flavour of pWhile, we now give programs modelling the examples presented in Section 1.1.
begin
a := c; # Alice’s money
b := 6 - c; # Bob’s money
while a > 0 && b > 0 do
  x ?= {-1,1};
  a := a + x;
  b := b - x
od;
stop
end

Figure 2.2: Game of chance

begin
# Throw dice
d1 ?= {1..6};
d2 ?= {1..6};
# Set the result
if d1==2 || d1==5 || d2==2 || d2==5
  then r := 0 # You win
else r := 1 # I win
fi;
stop
end

Figure 2.3: Two dice (2)

begin
# Randomly select the payer
b ?= {0,1,2,3};
# Coin tosses
for i:=0; i<3; i:=i+1 do
  c[i] ?= {0,1}
od;
# Announce bits publicly
for i:=0; i<3; i:=i+1 do
  if b != i
    then p[i] := c[i]^c[(i+1)%3]
  else p[i] := 1^(c[i]^c[(i+1)%3])
  fi
od;
stop
end

Figure 2.4: Dining Cryptographers

begin
# Pick winning door
d ?= {0,1,2};
# Pick guessed door
g ?= {0,1,2};
# Open empty door
o ?= {0,1,2};
while (o == g) || (o == d) do
  o := (o+1)%3;
o;
# Switch guess
g := (g+1)%3;
while (g == o) do
  g := (g+1)%3;
o;
stop; # looping
end

Figure 2.5: Monty Hall [4]
The program corresponding to Example 1.1.5 (Population expansion) is presented in Section 6.2.2, whereas the model for Example 1.1.4(1) is similar to the one shown in Figure 2.3.

### 2.4 Linear Operator Semantics

We now introduce Linear Operator Semantics of \texttt{pWhile} programs, which underlies the implementation of our compiler. We aim to give some intuition behind the main ideas of LOS which we hope will make it easier for the reader to understand the details of \texttt{pwc}'s implementation. We will make use of the following program $P$ throughout our presentation:

```plaintext
var
  x : \{0..4\};
  y : \{0,1\};
begin
  choose 1//3: x:=0 or 2//3: x:=1 ro;
  while x<4 do
    y?={0,1};
    x:=x+y;
  od;
stop;
end
```

We will analyse this program using probabilistic techniques, which will result in a discrete-time Markov chain generator corresponding to $P$.

#### 2.4.1 Probabilistic state

We begin by introducing the concept of state, which will give rise to states of a Markov chain corresponding to $P$. Intuitively, we may think of state as a mapping of variable names to values. In other words, it assigns a value to each variable in our program. For instance, $\{ x \mapsto 2, y \mapsto 1 \}$ is a valid state of $P$. In general, since number of variables is finite and we may order them according to the sequence in which they were declared in a program, state can be represented as an element of the Cartesian product\footnote{Cartesian product of two sets $A$ and $B$ is a set of pairs $(a, b)$ where $a \in A$ and $b \in B$. This can be easily generalised to any finite number of sets by replacing a pair with a tuple.}

$$\text{States} = \text{Values}_1 \times \text{Values}_2 \times \ldots \times \text{Values}_n$$

where $\text{Values}_i$ denotes the range of $i^{th}$ variable. For instance, states of $P$ are elements of $\{0, 1, 2, 3, 4\} \times \{0, 1\}$ and the state $\{ x \mapsto 2, y \mapsto 1 \}$ may be represented more concisely by a pair $(2, 1)$. We also note that the set of integers $\mathbb{Z}$ is totally ordered by the $\leq$ relation and we can define the order on the set
Table 2.1: Ordered states of $P$

<table>
<thead>
<tr>
<th>s_i</th>
<th>(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_1</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>s_2</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>s_3</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>s_4</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>s_5</td>
<td>(2, 0)</td>
</tr>
<tr>
<td>s_6</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>s_7</td>
<td>(3, 0)</td>
</tr>
<tr>
<td>s_8</td>
<td>(3, 1)</td>
</tr>
<tr>
<td>s_9</td>
<td>(4, 0)</td>
</tr>
<tr>
<td>s_{10}</td>
<td>(4, 1)</td>
</tr>
</tbody>
</table>

$\mathbb{B}$ of booleans by mapping $\texttt{false}$ to 0 and $\texttt{true}$ to 1 and using the order of $\mathbb{Z}$. Hence each set $\texttt{Values}$, is ordered. We may now use lexicographical ordering\(^3\) to specify the order on $\texttt{States}$. The ordered set of states of $P$ is given in Table 2.1.

We may now think of $\texttt{States}$ as the set of states of the Markov chain corresponding to $P$. We will later extend it to include the position in the execution of the program, but for now it is more convenient to consider $\texttt{States}$ as defined. We now define $\textit{probabilistic state}$ as a (discrete) random variable $S$ whose possible values are elements of $\texttt{States}$. We will call its probability distribution a $\textit{state vector}$.

In practice, we would like to be able to compute the distribution of $S$ given the distributions of variables efficiently. In some cases, we may use Kronecker product for that. In particular, we take advantage of the correspondence between distributions over $\texttt{States}$ and Kronecker product of distributions over $\texttt{Values}$.

The following example should make it clear.

$\textbf{Example 2.4.1.}$ Consider again the program $P$. Let $v_x$ denote the distribution of $x$ and $v_y$ the distribution of $y$. Suppose that before the first statement of the while loop $y$ takes value 1 and $x$ takes values 0 or 1, with equal probabilities. Therefore

\[
v_x = \left( \frac{1}{2}, \frac{1}{2}, 0, 0, 0 \right) \quad v_y = (0, 1)
\]

and

\[
v = (0, \frac{1}{2}, 0, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0) = v_x \otimes v_y
\]

is the state vector of that program at that point of execution. For example, $4^{th}$ entry of $v$ is the probability of state $s_4$. Upon executing

\(^3\)Given two ordered sets $A$ and $B$, the $\textit{lexicographical order}$ on the Cartesian product $A \times B$ is defined as $(a, b) \leq (a', b')$ iff $a < a'$ or $a = a'$ and $b \leq b'$.
\[ y \equiv \{0, 1\}; \]

which assigns 0 or 1 (with equal probabilities) to \( y \), the distribution of \( y \) becomes

\[ \mathbf{v}_y = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \]

and the state vector is

\[ \mathbf{v} = \left( \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, 0, 0, 0, 0, 0, 0 \right) = \mathbf{v}_x \otimes \mathbf{v}_y \]

It might seem at this point that our representation of probabilistic state as a vector with ten entries is redundant. One may argue that each entry of \( \mathbf{v} \) can be recovered by multiplying corresponding entries of \( \mathbf{v}_x \) and \( \mathbf{v}_y \), as suggested by the above equalities. Therefore, it might be tempting to drop \( \mathbf{v} \) and only store the distributions of individual variables, which certainly saves space. Consider what happens to the state vector upon executing the second statement in the loop, i.e.

\[ \mathbf{x} := \mathbf{x} + \mathbf{y}; \]

We recall that both \( \mathbf{x} \) and \( \mathbf{y} \) take on values 0 and 1 with equal probabilities. Hence after executing that line, four states are possible – \( \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 1 \rangle \) and the state vector is

\[ \mathbf{v} = \left( \frac{1}{4}, 0, \frac{1}{4}, \frac{1}{4}, 0, \frac{1}{4}, 0, 0, 0, 0 \right) \]

On the other hand, the individual distributions of \( \mathbf{x} \) and \( \mathbf{y} \) are

\[ \mathbf{v}_x = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \]

\[ \mathbf{v}_y = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \]

and so

\[ \mathbf{v} \neq \mathbf{v}_x \otimes \mathbf{v}_y \]

In other words, the entries of \( \mathbf{v} \) can no longer be recovered from the entries of \( \mathbf{v}_x \) and \( \mathbf{v}_y \). The reason for that is that \( \mathbf{x} \) and \( \mathbf{y} \) are now dependent. We conclude that probabilistic state must indeed be stored as a separate vector with the number of entries equal to the number of different states.

### 2.4.2 Labels

We begin analysing any program by detecting its blocks, and labelling them. Blocks are the atomic executable statements and each is assigned a unique label. The labelled version of \( P \) is
Consider the `choose` statement as an example. We may view its execution as a two-stage process: first, the probabilistic choice is made to decide which of the two statements to execute, followed by the execution of the chosen statement. We therefore assign its blocks as shown above. We can now view any execution of $P$ as “jumping” between blocks and every such run may be represented uniquely as a sequence of labels. Due to the probabilistic nature of `pWhile`, several orders are possible in general, and to each of them we can associate probability of it taking place. Of course, there are certain restrictions on the possible sequences produced. For instance, 1 must be followed by either 2 or 3 - reflecting the fact that executing `choose` block is always followed by running one of the two alternatives. Similarly, 5 must always be followed by 6. Finally, note that due to the semantics of a `stop` statement, all sequences of labels corresponding to executions of $P$ will be infinite. We will make the above remarks more precise in Section 2.4.3.

In general, let $Stmt$ and $LStmt$ denote the set of statements and labelled statements respectively, and $Lab$ the set of labels. We first define an auxiliary function $newlabel$

$$newlabel : \text{Void} \rightarrow \text{Lab}$$

which returns a different label each time it is called (easiest way to achieve that is clearly to return 1 on the first call, and subsequent integers on subsequent calls). We may then define the function $label$ which assigns labels to a given statement

$$label : Stmt \rightarrow LStmt$$
in the following way

\[
\begin{align*}
\text{label}(\text{skip}) & = [\text{skip}]^\ell \\
\text{label}(\text{stop}) & = [\text{stop}]^\ell \\
\text{label}(v := e) & = [v := e]^\ell \\
\text{label}(v ?= e) & = [v ?= e]^\ell \\
\text{label}(S_1; S_2) & = \text{label}(S_1); \text{label}(S_2) \\
\text{label}(\text{choose } p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}) & = [\text{choose}]^\ell \ p_1 : \text{label}(S_1) \text{ or } p_2 : \text{label}(S_2) \text{ ro} \\
\text{label}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}) & = \text{if } [b]^\ell \text{ then } \text{label}(S_1) \text{ else } \text{label}(S_2) \text{ fi} \\
\text{label}(\text{while } b \text{ do } S \text{ od}) & = \text{while } [b]^\ell \text{ do } \text{label}(S) \text{ od}
\end{align*}
\]

where

\[ \ell = \text{newlabel()} \]

in each of the above. One can now easily check that the labelling of \( P \) presented above is obtained by applying function \( \text{label} \) to \( P \).

### 2.4.3 Control flow

Once we labelled the program, we are ready to compute its flow. Intuitively, flow represents the possible transitions between blocks and probabilities associated with them (compare this with transitions between states of Markov Chain). More formally, flow \( \mathcal{F} \) is a set of triples \( \langle \ell_i, p_{ij}, \ell_j \rangle \), where each such triple reflects the fact that control passes with probability \( p_{ij} \) from block \( b_i \) to block \( b_j \) [4]. For example, the flow of \( P \) is

\[
\begin{align*}
\text{flow}(P) & = \{\langle 1, \frac{1}{3}, 2 \rangle, \langle 1, \frac{2}{3}, 3 \rangle, \langle 2, 1, 4 \rangle, \langle 3, 1, 4 \rangle, \langle 4, 1, \underline{5} \rangle\} \\
& \quad \cup \{\langle 4, 1, 7 \rangle, \langle 5, 1, 6 \rangle, \langle 6, 1, 4 \rangle, \langle 7, 1, 7 \rangle\}.
\end{align*}
\]

Note that it is our convention to use underline to differentiate between the two branches of a conditional. In the above, block \( b_4 \) is a conditional and depending on its truth value, the control passes to blocks \( b_5 \) or \( b_7 \). Therefore, we underline the 5 to convey that if the condition is true, control passes to block \( b_5 \). Note also that the probability \( p_{ij} \) is almost always 1, except for the triples corresponding to the \textbf{choose} block.

We now give a formal method to compute flow of any program, which requires a little groundwork first. Recall \textbf{LStmt} denotes the set of labelled statements and \textbf{Lab} the set of labels. We let \( \mathcal{P}(\text{Lab}) \) denote the power set of \textbf{Lab}, i.e. the set of subsets of \textbf{Lab}. We may now define two auxiliary operations \textit{init} and \textit{final}

\[
\begin{align*}
\textit{init} : \text{LStmt} & \rightarrow \text{Lab} \\
\textit{final} : \text{LStmt} & \rightarrow \mathcal{P}(\text{Lab})
\end{align*}
\]

20
which return the initial label and the final labels of a statement respectively (note that any statement has a single entry but may have multiple exists, e.g. if or choose statements). We define them in the following way [4]:

\[
\begin{align*}
\text{init}([\text{skip}]^\ell) &= \ell \\
\text{init}([\text{stop}]^\ell) &= \ell \\
\text{init}([v := e]^\ell) &= \ell \\
\text{init}([v \neq e]^\ell) &= \ell \\
\text{init}(S_1; S_2) &= \text{init}(S_1) \\
\text{init}([\text{choose}]^\ell p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}) &= \ell \\
\text{init}(\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= \ell \\
\text{init}(\text{while } [b]^\ell \text{ do } S \text{ od}) &= \ell
\end{align*}
\]

and

\[
\begin{align*}
\text{final}([\text{skip}]^\ell) &= \{\ell\} \\
\text{final}([\text{stop}]^\ell) &= \{\ell\} \\
\text{final}([v := e]^\ell) &= \{\ell\} \\
\text{final}([v \neq e]^\ell) &= \{\ell\} \\
\text{final}(S_1; S_2) &= \text{final}(S_2) \\
\text{final}([\text{choose}]^\ell p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}) &= \text{final}(S_1) \cup \text{final}(S_2) \\
\text{final}(\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= \text{final}(S_1) \cup \text{final}(S_2) \\
\text{final}(\text{while } [b]^\ell \text{ do } S \text{ od}) &= \{\ell\}
\end{align*}
\]

We are now ready to define the main function \( \text{flow} \):

\[
\text{flow} : \text{LStmt} \rightarrow P(\text{Lab} \times [0,1] \times \text{Lab})
\]

which, given a statement (i.e. a program) returns the set of triples representing
its control flow \([4]\):

\[
\text{flow}([\text{skip}]^\ell) = \emptyset \\
\text{flow}([\text{stop}]^\ell) = \{(\ell, 1, \ell)\} \\
\text{flow}([v := e]^\ell) = \emptyset \\
\text{flow}([v ?= e]^\ell) = \emptyset \\
\text{flow}(S_1; S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
\{((\ell, 1, \text{init}(S_2)) \mid \ell \in \text{final}(S_1)\} \\
\text{flow}([\text{choose}]^\ell p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
\{((\ell, p_1, \text{init}(S_1)), (\ell, p_2, \text{init}(S_2)))\} \\
\text{flow}([\text{if}]^\ell [b] \text{ then } S_1 \text{ else } S_2 [\text{fi}]) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
\{((\ell, 1, \text{init}(S_2)), (\ell, 1, \text{init}(S_2)))\} \\
\text{flow}([\text{while}]^\ell [b] \text{ do } S \text{ od}) = \text{flow}(S) \cup \\
\{((\ell, 1, \text{init}(S)), (\ell', 1, \ell) \mid \ell' \in \text{final}(S))\}
\]

The reader may again check that the flow of \(P\) given above is consistent with the definition just presented.

### 2.4.4 Local operators

With flow in hand, we may begin giving shape to our ultimate goal, i.e. the DTMC generator. The basic idea is to consider each flow triple separately and produce a local operator (which is a matrix) corresponding to it, followed by a clever use of Kronecker product to glue them all together, resulting in a Markov chain generator. We now present how the local operators are computed.

Recall that in Section 2.4.1, we introduced the concept of state (respectively probabilistic state) as an element of (respectively random variable over) the set \(\text{States}\), defined as

\[
\text{States} = \text{Values}_1 \times \text{Values}_2 \times \ldots \times \text{Values}_n
\]

where \(\text{Values}_i\) denotes the range of \(i^{th}\) variable. By assumption, \(\text{Values}_i\) is a finite set for any \(i\), and hence \(\text{States}\) is finite too. We compute its size as

\[
n = |\text{States}| = |\text{Values}_1| \cdot |\text{Values}_2| \cdot \ldots \cdot |\text{Values}_n|
\]

which gives the number of unique configurations of variable values in any program. We recall that distribution of probabilistic state \(S\) may be represented by a vector \(s\), called the state vector. Since the size of the distribution vector of a discrete random variable is the number of its possible values, the length of \(s\) is \(n\).

Each flow triple represents a transition from one block to another. Hence the matrix corresponding to such triple should reflect the effects on state of
executing the first block of the triple. More precisely, we would like to find a
matrix \( \mathbf{T} \), such that \( \mathbf{s} \cdot \mathbf{T} \) is equal to the state vector after executing the block.
Since \( \mathbf{s} \) has length \( n \), the dimensions of matrix operators must be \( n \times n \). Their
entries clearly depend on the block executed, however they can be classified into
three generic categories, which we now present.

**Identity Operator**

We start with the most straightforward operator, which corresponds to \texttt{skip},
\texttt{stop} and \texttt{choose} blocks. Since executing them does not modify the state at all,
they give rise to the identity operator \( \mathbf{I} \), which is a \( n \times n \) identity matrix.

**Update Operator**

Next we consider an operator corresponding to the assignment statement. We
call it an update operator, since it updates the (probabilistic) state. Before we
define it formally, we explain the intuition behind it, supporting ourselves with
an example. For that, we return to program \( P \) introduced at the start of this
section. We identify its two variables, \( x \) and \( y \), with ranges \( \{0, 1, 2, 3, 4\} \) and
\( \{0, 1\} \) respectively. Hence there are \( n = 5 \cdot 2 = 10 \) different states possible. The
order of the two variables is given by the order in which their declarations appear
in the program. The update operator corresponding to block \( b_2 \) (equivalently, to
flow triple \( \langle 2, 1, 4 \rangle \)), denoted \( \mathbf{U}(x \leftarrow 0) \), is

\[
\begin{pmatrix}
    s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \\
   1 & . & . & . & . & . & . & . & . & . \\
   . & 1 & . & . & . & . & . & . & . & . \\
   1 & . & . & . & . & . & . & . & . & . \\
   . & 1 & . & . & . & . & . & . & . & . \\
   1 & . & . & . & . & . & . & . & . & . \\
   . & 1 & . & . & . & . & . & . & . & . \\
   1 & . & . & . & . & . & . & . & . & . \\
   . & 1 & . & . & . & . & . & . & . & . \\
   . & . & . & . & . & . & . & . & . & . \\
\end{pmatrix}
\]

Note that we represent zeros by dots to improve clarity. It is important now to
recall that we have equipped \texttt{States} with lexicographical order. For our program
\( P \) it means that state \( \langle 0, 0 \rangle \) (i.e. \( \{ x \mapsto 0, y \mapsto 0 \} \)) is the first state, \( \langle 0, 1 \rangle \) the
second etc., up until \( \langle 4, 1 \rangle \) being the last state. We recall Table 2.1 which shows
the order explicitly.

In this setting, the \( ij \) – entry of \( \mathbf{U}(x \leftarrow 0) \) corresponds to probability of
moving to state \( s_j \) after executing assignment under consideration, assuming that
the computation was in state \( s_i \) before the assignment. Since it is deterministic,
all the probabilities in the above matrix are either 1 or 0. Moreover, there is
only one nonzero entry in each row, which makes it stochastic. As expected, the only two states which are possible after executing $x:=0$ are $s_1$ and $s_2$, which are the only states in which $x$ has value 0.

An attentive reader might have noticed that the matrix above can be expressed as a Kronecker product in the following way

$$U(x \leftarrow 0) = \begin{pmatrix} 1 & \ldots & \ldots & \ldots & 1 & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \ldots & \ldots & \ldots & 1 & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \ldots & \ldots & \ldots & 1 & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \ldots & \ldots & \ldots & 1 & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Of course, this is not a coincidence. In fact, we have already seen something similar in Example 2.4.1, where the distribution of two variables was expressed as a Kronecker product of individual distributions. We also observed there that such a “factorisation” is possible if and only if the variables are independent. Accordingly, an update operator acting on all variables of the program can be expressed as a Kronecker product of operators acting on each variable individually if and only if the assigned value (i.e. the RHS of the assignment) is a constant. To illustrate this, we now give an update operator $U(x \leftarrow x + y)$ corresponding to block $b_6$ (equivalently, flow triple $(6, 1, 4)$):


Clearly, the matrix above is not expressible as a Kronecker product of $5 \times 5$ and $2 \times 2$ matrices. It also shows another interesting feature of our semantics - since each variable has a finite range, we need to address the issue of assigned value falling out of its range. The simplest solution is to treat it “literally”: since we defined the update operator to be a matrix whose $ij$-entry is the probability of moving from state $s_i$ to $s_j$ upon executing the assignment, if the
assignment makes one of the variables fall out of range, none of the well-defined states is reached and we get a row of zeros. This is the case for the last row of $U(x \leftarrow x + y)$. In fact, in this case it does not matter what the last row is, since from the code we see that state $s_{10}$ is not reachable inside the loop. However, in some situations it might not be desirable to let the update operator (and consequently also the generator of DTMC, as we will see shortly) lose the stochastic property, as it may in turn lead to the state vector losing its stochastic property. One way to address it is briefly described in Section 7.1.

With are now in a position to consider the operator associated to random assignment. It should not come as a surprise that it is very closely related to the update operator. In fact it is simply a sum of update operators, multiplied by an appropriate fraction. For instance, the operator corresponding to block $b_5$ ([y ?= \{0,1\}] is

$$
\frac{1}{2} U(y \leftarrow 0) + \frac{1}{2} U(y \leftarrow 1) = \\
\begin{pmatrix}
s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \\
\frac{1}{2} & \frac{1}{2} & . & . & . & . & . & . & . & . \\
\frac{1}{2} & \frac{1}{2} & . & . & . & . & . & . & . & . \\
. & . & \frac{1}{2} & \frac{1}{2} & . & . & . & . & . & . \\
. & . & \frac{1}{2} & \frac{1}{2} & . & . & . & . & . & . \\
. & . & . & . & \frac{1}{2} & \frac{1}{2} & . & . & . & . \\
. & . & . & . & \frac{1}{2} & \frac{1}{2} & . & . & . & . \\
. & . & . & . & . & . & \frac{1}{2} & \frac{1}{2} & . & . \\
. & . & . & . & . & . & \frac{1}{2} & \frac{1}{2} & . & . \\
. & . & . & . & . & . & . & . & \frac{1}{2} & \frac{1}{2} \\
\end{pmatrix}
$$

reflecting the fact that $y$ gets assigned 0 or 1 with equal probabilities. It is an easy exercise to generalise it to an arbitrary random assignment.

**Filter Operator**

The last block for which we have to define a corresponding operator is the “conditional”, which appears as part of if and while statements. Recall the while loop in our program:

```
while [x<4] \^4 do
    [y?=\{0,1\}] \^5 ;
    [x:=x+1] \^6 ;
od;
```

The block $b_4$ is a “conditional” block and we associate two operators with it - corresponding to the binary expression being true or false. Those two operators are in a way complements of each other - having computed one, we can easily get the other. In general, a filter operator takes any binary expression as argument and produces a matrix, which filters the state vector to allow only states that
make the binary expression true. The most straightforward filter operators are
\( P(\text{true}) \) and \( P(\text{false}) \). The former is an identity matrix, since \textit{true} is true in
any state, while the latter is the zero matrix, since \textit{false} is false in any state.
We also see from that example what notion of complementarity are we dealing
with - for any binary expression \( b \)

\[
P(b) + P(\neg b) = I
\]

In general, let \( b \) be a binary expression valid in the scope of some program
\( P \) (by which we mean that any variables appearing in \( b \) are declared in \( P \)) and
\( s_1, s_2, \ldots, s_n \) be ordered states of \( P \). We introduce the notation \( b|s_n \) to denote \( b \)
evaluated in state \( s_n \). Then the filter operator \( P(b) \) is an \( n \times n \) diagonal matrix
with only zeros and ones on the diagonal, satisfying the following property:
\( P(b)_{ii} = 1 \) if and only if \( b|s_i \).

As an example, consider the filter operators corresponding to block \( b_4 \) of the
\textbf{while} loop recalled above. We have

\[
P(x < 4) = \begin{pmatrix}
1 & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . \\
. & . & 1 & . & . & . & . \\
. & . & . & 1 & . & . & . \\
. & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
\end{pmatrix}
\]

\[
P(\neg(x < 4)) = \begin{pmatrix}
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
\end{pmatrix}
\]

A helpful way to get some intuition behind the filter operator is to think
about the deterministic program, in which the state vector \( s \) is going to have a
single entry equal to 1 and all other entries 0, at all times during the execution
of the program. On entering the \textbf{if} statement, this single state will either make
the condition \( b \) true, or false. In this case, we have (without loss of generality)

\[
s \cdot P(b) = s \\
s \cdot P(\neg b) = 0
\]
which represents the fact that the control enters one of the branches of the if statement. The probabilistic case is a generalisation of this behaviour.

**Formal definitions**

Now that we have some intuition behind all the matrix operators used in Linear Operator Semantics, we give their formal definitions. While it is not strictly necessary, we believe it is helpful to express the above ideas rigourously, using mathematical formulas. Such presentation is more concise, avoids any understatements, and serves as a good reference. Moreover, it helps with implementation and provides a good indication of where the inefficiencies may lie. The definitions we give are inspired by the ones given by H. Wiklicky, A. Di Pierro and C. Hankin [4], however we diverge from them slightly.

We begin with a technical note: it is often more convenient for us to deal with indexes into variable’s range, rather than its values. For instance, if a variable \( x \) has range \( \{4, 5, 6, 7\} \), then assigning value 6 to it produces update operator with third column consisting of ones, and all other entries zero. Importantly, what matters from the point of view of the resulting operator is the index of the assigned value (6 in this case) in \( x \)’s range. Therefore, to keep the notation simple, we use \( x \leftarrow c \) to denote the operator which assigns to \( x \) the \( c^{th} \) value in its range. Similarly, \( x|_{s_i} \) is assumed to return the index of \( x \)’s value in state \( s_i \), rather than the value itself. Finally, for the operator \( \mathbf{U}(x \leftarrow e|_{s_i}) \), expression \( e \) is evaluated in the context of \( x \), which means that it returns the index into range of \( x \).

| \( (\mathbf{P}(s_k))_{ij} \) | \= | \begin{cases} 1 & \text{if } i = k = j \\ 0 & \text{otherwise} \end{cases} | \\
| \mathbf{P}(b) \= | \sum_i b|_{s_i} \cdot \mathbf{P}(s_i) |

Table 2.2: Filter Operators

The filter operator \( \mathbf{P}(b) \), defined in Table 2.2, is a sum of state filter operators \( \mathbf{P}(s_i) \), but only those states are included in the sum which make \( b \) true (note the implicit conversion of boolean \( b|_{s_i} \) to 1 when it is true, 0 otherwise). Note that every state operator \( \mathbf{P}(s_i) \) is simply a matrix with 1 in the \( i^{th} \) diagonal entry and zeros elsewhere.

Similarly, Table 2.3 defines update operators. We have seen in Section 2.4.4 that when assigned expression is a constant, then the operator \( \mathbf{U}(v_k \leftarrow c) \) may be expressed as a Kronecker product of operators acting on individual variables. The update operator acting on \( v_k \) individually is a matrix with \( c^{th} \) column of ones and all other entries zero. Since an assignment to \( v_k \) does not modify any other
\[(U(c))_{ij} = \begin{cases} 1 & \text{if } j = c \\ 0 & \text{otherwise.} \end{cases} \]

\[U(v_k \leftarrow c) = \bigotimes_{i=1}^{k-1} I \otimes U(c) \otimes \bigotimes_{i=k+1}^{v} I\]

\[U(v_k \leftarrow e) = \sum_i P(s_i)U(v_k \leftarrow e|_{s_i})\]

| Table 2.3: Update Operators |

variables, the other operators making up the Kronecker product are identity operators. Finally, the operator \(U(v_k \leftarrow e)\) is defined in terms of the state filter operator \(P(s_i)\) and \(U(v_k \leftarrow e|_{s_i})\). In particular, expression \(e\) is evaluated in every state to produce a constant, which is then assigned to \(v_k\) using already defined constant update operator. To limit the scope of the assignment to the particular state in which the expression was evaluated, we pre-multiply each such constant update operator by \(P(s_i)\).

### 2.4.5 Global operators and DTMC generator

The last step in computing the Markov chain generator is combining the operators corresponding to all the blocks in the program. So far we have considered state to be the Cartesian product of ranges of variables. However, there is one more component which is part of the state of execution of a probabilistic program at any given time, namely the program counter. Taking advantage of the labelling already introduced, the program counter may be represented by a label of the next block to be executed. We therefore extend our definition of state in the following way:

\[\text{States} = \text{Values}_1 \times \text{Values}_2 \times \ldots \times \text{Values}_n \times \text{Lab}\]

where, as before, \(\text{Values}_i\) denotes the range of \(i^{th}\) variable.

It is often useful to refer to the “old” state as variable state and set

\[\text{VarStates} = \text{Values}_1 \times \text{Values}_2 \times \ldots \times \text{Values}_n\]

so that

\[\text{States} = \text{VarStates} \times \text{Lab}\]

We will also call the vector representing the distribution of the program counter a block vector. Provided distributions of \(\text{VarStates}\) and \(\text{Lab}\) are independent, we may represent the probabilistic state \(s\) as

\[s = \text{vs} \otimes b\]

28
where \( \mathbf{v} \) is a variable state vector and \( \mathbf{b} \) is a block vector. Defined in that way, \( \text{States} \) is now the set of states of a discrete-time Markov chain corresponding to the program. We now let

\[
    n = |\text{States}| = |\text{Values}_1| \cdot |\text{Values}_2| \cdot \ldots \cdot |\text{Values}_n| \cdot |\text{Lab}|
\]

denote the number states.

We therefore expect the Markov chain generator to be an \( n \times n \) matrix. To achieve that, we define matrix units \( E(i,j) \) as

\[
    E(m, n)_{ij} = \begin{cases} 1 & \text{if } m = i \wedge n = j \\ 0 & \text{otherwise.} \end{cases}
\]

We may now introduce global operators \( T(\ell_i, \ell_j) \), which essentially put the local operators in the context of the whole program. Each global operator is of the form \( \mathbf{N} \otimes E(\ell_i, \ell_j) \) where the first factor \( \mathbf{N} \) represents a local operator while the second factor realises the transfer of control from label \( \ell_i \) to label \( \ell_j \) \cite{4}. Table 2.4 shows global operators corresponding to labelled blocks, bringing together everything we have presented so far.

| \( T(\ell_1, \ell_2) \) | = | \( T(\ell, \ell) \) | = | \( T(\ell_1, \ell_2) \) | = | \( T(\ell, \ell_k) \) | = | \( T(\ell, \ell_f) \) | = |
|---|---|---|---|---|---|
| \( I \otimes E(\ell_1, \ell_2) \) | \( I \otimes E(\ell, \ell) \) | \( U(v \leftarrow c) \otimes E(\ell_1, \ell_2) \) | \( I \otimes E(\ell, \ell_k) \) | \( P(b = \text{true}) \otimes E(\ell, \ell_t) \) | \( P(b = \text{false}) \otimes E(\ell, \ell_f) \) |

for \( \text{[skip]}^{\ell_i} \)

for \( \text{[stop]}^{\ell} \)

for \( \text{[v := e]}^{\ell_1} \)

for \( \text{[choose]}^{\ell} \)

for \( \text{[b]}^{\ell} \)

for \( \text{[b]}^{\ell} \)

Table 2.4: Global Operators \cite{4}

We are finally ready to define the DTMC generator \( T(P) \) corresponding to any probabilistic program \( P \) \cite{4}:

\[
    T(P) = \sum_{(i,p_{ij},j) \in F(P)} p_{ij} \cdot T(\ell_i, \ell_j).
\]

It consists of global operators corresponding to each flow triple, weighted according to the probability associated to this triple.

It might not be obvious at first why this works. The reason is the beauty of Kronecker product, in particular the mixed-product property mentioned in Section 2.2.3. The following example should clear it up.
Example 2.4.2. Consider an abstract program $P$ with three blocks and flow

$$F = \{(1,1,2), (2,1,3), (3,1,3)\}$$

Suppose the local operators corresponding to blocks $b_1$, $b_2$ and $b_3$ are given by

$$L(b_1) = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & 0 \end{pmatrix}$$

$$L(b_2) = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ 0 & 1 \end{pmatrix}$$

$$L(b_3) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Then the DTMC generator $T(P)$ is

$$T(P) = L(b_1) \otimes E(1,2) + L(b_2) \otimes E(2,3) + L(b_3) \otimes E(3,3)$$

Hence we see that $T(P)$ contains “stretched and translated” copies of the local operators. Suppose now that initial state vector is

$$s_0 = (1,0,0,0,0) = (1,0) \otimes (1,0,0) = vs \otimes b$$

where $vs$ is the distribution of variable state and $b$ is the distribution of program counter. Let us now “execute” one step of the abstract program $P$:

$$s_1 = s_0 \cdot T(P)$$

$$= s_0 \cdot (L(b_1) \otimes E(1,2)) + s_0 \cdot (L(b_2) \otimes E(2,3)) + s_0 \cdot (L(b_3) \otimes E(3,3))$$

$$= s_0 \cdot (L(b_1) \otimes E(1,2))$$

$$= (vs \otimes b) \cdot (L(b_1) \otimes E(1,2))$$

$$= (0, \frac{1}{2}, 0, 0, \frac{1}{2}, 0)$$

$$= (vs \cdot L(b_1)) \otimes (b \cdot E(1,2))$$

Of course, it is not a coincidence that in the above

$$(vs \otimes b) \cdot (L(b_1) \otimes E(1,2)) = (vs \cdot L(b_1)) \otimes (b \cdot E(1,2))$$

This is in fact the mixed-product property of Kronecker product. We see above how the local operator acts on the variable state vector and the transfer operator $E(\ell_i, \ell_j)$ acts on the block vector.
We conclude by computing the generator of program $P$ introduced at the start of this section. Recall that $P$ is

\begin{verbatim}
var
  x : {0..4};
  y : {0,1};
begin
  [choose] 1//3:[x:=0] or 2//3:[x:=1] \; //
  [y?={0,1}] \;
  [x:=x+1] \;
  od;
  [stop];
end
\end{verbatim}

We computed the flow of $P$ in Section 2.4.3 as

\[
\text{flow}(P) = \{ (1, \frac{1}{3}, 2), (1, \frac{2}{3}, 3), (2, 1, 4), (3, 1, 4), (4, 1, 5) \} \\
\cup \{ (4, 1, 7), (5, 1, 6), (6, 1, 4), (7, 1, 7) \}.
\]

The local operators corresponding to blocks $b_1$ and $b_7$ are identity operators. We computed the update operator $U(x \leftarrow 0)$ corresponding to block $b_2$ in Section 2.4.4. One can easily compute the operator $U(x \leftarrow 1)$ corresponding to block $b_3$ using similar technique. Filter operators associated to the conditional block $b_4$ are $P(x < 4)$ and $P(\neg(x < 4))$ given in that same section, along with operators corresponding to blocks $b_5$ and $b_6$. We now give the global operators of $P$:

\[
\begin{align*}
T(1,2) &= I \otimes E(1,2) \\
T(1,3) &= I \otimes E(1,3) \\
T(2,4) &= U(x \leftarrow 0) \otimes E(2,4) \\
T(3,4) &= U(x \leftarrow 1) \otimes E(3,4) \\
T(4,5) &= P(x < 4) \otimes E(4,5) \\
T(5,6) &= \left( \frac{1}{2} U(y \leftarrow 0) + \frac{1}{2} U(y \leftarrow 1) \right) \otimes E(5,6) \\
T(6,4) &= U(x \leftarrow x + y) \otimes E(6,4) \\
T(4,7) &= P(\neg(x < 4)) \otimes E(4,7) \\
T(7,7) &= I \otimes E(7,7)
\end{align*}
\]

Hence the DTMC generator corresponding to $P$ is a $70 \times 70$ matrix

\[
T(P) = T(1,2) + T(1,3) + T(2,4) + T(3,4) + T(4,5) + T(5,6) + T(6,4) + T(4,7) + T(7,7)
\]
Chapter 3

Design & Language Extension

This chapter presents the high-level structure of pwc and the theoretical part of the project – extending pWhile and LOS.

3.1 Design

The flowchart below shows the main components of our tool, along with the sequence of steps performed as part of the compilation process.
As in any compiler, the input program $P$ is first processed by a parser, which breaks it up into tokens, thereby verifying its syntactic correctness, and produces its structured representation using data types which we defined. In particular, the result of parsing is a pair, whose first element is a list of declarations and the second is the syntax tree of the program. Each declaration is a pair $(v, m)$, where $v$ is the name of the declared variable and $m$ is its meta data, which contains information such as its type or range. It might seem unnecessary at this point, since we only introduced primitive values. However, we will soon bring in constants and arrays, which will make the meta data needful. The OCaml definitions of the types described above are presented in Appendix A.1. The syntax tree of the program consists of different kinds of statements, some of which additionally contain arithmetic and binary expressions. Both of these types closely reflect the syntax of $p$While and their definitions are again given in Appendix A.1 (Note that they contain more language constructs than we introduced so far. The extra syntax is presented in Section 3.2).

Upon successful parsing, the program is analysed semantically. The details of this phase of compilation are described in Section 4.3.2. If the semantic verification succeeds, the compiler begins to analyse $P$ from the point of view of Linear Operator Semantics. The process closely resembles our course of action from Section 2.4. First, the syntax tree is passed to the Label module, which returns the labelled version of the supplied program. It is in turn passed to two other modules: first to the Block module, which computes the blocks of $P$, and then to the Flow module which computes the flow of $P$. The final step is the generation of a Julia file which defines the LOS semantics of the program, i.e. all the operators described in Sections 2.4.4 and 2.4.5. We often refer to this part of our compiler as “Julia back end” and we defer analysis of its implementation to Section 4.4.

### 3.2 Language extensions

Chronologically, the first part of the project was to add several new language constructs to the $p$While language. The purpose of this exercise was two-fold: (i) it clearly adds some value to the tool itself; using new syntax, programmers can write more concise and clearer programs, (ii) it enabled us to familiarise ourselves with the existing code and the details of LOS. For each new statement introduced, we updated the Linear Operator Semantics accordingly. We now list the language extensions and the corresponding updates to LOS.

#### 3.2.1 Arrays

One of the fundamental programming concepts missing in $p$While, as introduced in Section 2.3 is an array. Even though any array may be imitated by a set of variables, it is very inconvenient and sometimes (when the array is big) impracticable.
We begin by presenting syntax extensions to \texttt{pWhile}. Just like usual variables, before being used, an array must be declared. To declare an array \texttt{arr}, of size \( n \) with each element in range \( r \) one would include

\[
\texttt{arr}[n] : r
\]

in the declaration part of the \texttt{pWhile} program.

We use the convention that arrays are indexed starting at 0. To assign to the \( i^{th} \) element (where \( i \) is an arithmetic expression) of array \texttt{arr} expression \( a \) or range \( r \) respectively, one would use one of the following statements:

\[
\texttt{arr}[i] := a \\
\texttt{arr}[i] \leftarrow r
\]

Similarly, we extend the syntax of arithmetic expressions to allow array references be part of them. Hence

\[
\texttt{arr}[i]
\]

where \( \texttt{arr} \) is an array declared in the program and \( i \) is an arithmetic expression is now a valid arithmetic expression.

Finally, we give an example \texttt{pWhile} program that uses an array.

\textbf{Example 3.2.1.} The following program produces an array which represents a number between 0 and 7 in binary format.

\begin{verbatim}
var
  bin[3] : \{0,1\};
begin
  bin[0]:=\{0,1\};
  bin[1]:=\{0,1\};
  bin[2]:=\{0,1\};
  stop;
end
\end{verbatim}

What if we wanted to represent a number between 0 and 1023 in binary using an array? The above approach certainly does not scale well. We could use a \texttt{while} loop to iterate over the array. However, the most natural solution for most programmers would probably be to use a \texttt{for} loop, which, as it stands, is not part of \texttt{pWhile} syntax.

\subsection*{3.2.2 For loop}

The next statement we introduce is therefore \texttt{for} loop. The \texttt{pWhile} syntax corresponding to it is

\begin{verbatim}
  for \( S_1 \) ; \( b \) ; \( S_2 \) do \( S \) od
\end{verbatim}

Its semantics is as follows:
- $S_1$ is the loop initialisation and is executed before the first iteration of the loop. Usually it’s a simple assignment, such as $i:=0$, however in principle we allow an arbitrary statement here.

- $b$ is the binary expression, which we refer to as loop test, and is executed before each iteration of the loop. If it evaluates to true, loop enters another iteration, otherwise it exits.

- $S_2$ is the loop update and is executed after each iteration of the loop. Again, it will usually be a simple assignment, such as $i:=i+1$, however we allow an arbitrary statement.

- $S$ is the body of the loop and is executed at every iteration of the loop

We now extend Linear Operator Semantics to support our newly introduced construct. The first step is to extend the label function:

$$\text{label}(\text{for } S_1 ; b ; S_2 \text{ do } S \text{ od}) = \text{for } \text{label}(S_1) ; [b]^{\ell} ; \text{label}(S_2) \text{ do } \text{label}(S) \text{ od}$$

where

$$\ell = \text{newlabel}()$$

After that, we update the init, final and flow functions:

$$\text{init}(\text{for } S_1 ; [b]^{\ell} ; S_2 \text{ do } S \text{ od}) = \text{init}(S_1)$$

$$\text{final}(\text{for } S_1 ; [b]^{\ell} ; S_2 \text{ do } S \text{ od}) = \{\ell\}$$

$$\text{flow}(\text{for } S_1 ; [b]^{\ell} ; S_2 \text{ do } S \text{ od}) = \text{flow}(S) \cup \text{flow}(S_1) \cup \text{flow}(S_2)$$

$$\cup \{(\ell',1,\ell) \mid \ell' \in \text{final}(S_1)\} \cup$$

$$\cup \{(\ell,1,\text{init}(S))\} \cup$$

$$\cup \{(\ell',1,\text{init}(S_2)) \mid \ell' \in \text{final}(S)\} \cup$$

$$\cup \{(\ell',1,\ell) \mid \ell' \in \text{final}(S_2)\} \cup \{\ell\} \cup \{\ell,1,\ell\} \cup \{\ell,1,\text{init}(S)\} \cup \{\ell,1,\text{init}(S_2)\}$$

We conclude by using for loop to write a program which computes a random binary number between 0 and 1023.

**Example 3.2.1.** Consider the following, labelled program $P$:

```plaintext
var
  i : {0..10};
  bin[10] : {0,1};
begin
  for [i:=0] \ {i<10} \ {i:=i+1} do
    [bin[i]?={0,1}];
  od;
  # 'bin' now contains a random
  # binary number in {0..1023}
[stop] ;
end
```

35
The flow of this program is given by:

\[ \text{flow}(P) = \{\langle 1, 1, 2 \rangle, \langle 2, 1, 4 \rangle, \langle 4, 1, 3 \rangle, \langle 3, 1, 2 \rangle, \langle 2, 1, 5 \rangle, \langle 5, 1, 5 \rangle\} \]

### 3.2.3 Case statement

Sometimes we want to write code that takes different actions depending on the value of a variable. We could achieve that using a nested if else statement, however arguably, the more elegant solution is to use the case (also known as switch) statement. To employ it in a pWhile program one would write

```pwhile
case a cases default esac
```

where \(a\) is an arithmetic expression, \(cases\) is one or more statements of the form

\[ \text{of } n: \ S \]

with \(n\) being an integer, \(S\) an arbitrary statement, and \(default\) stands for

\[ \text{default: } S \]

where, again, \(S\) is an arbitrary statement. The semantics of the case statement is the following:

1. The control flow enters the case statement, in particular the first case.

2. For each case, \(a\) is evaluated and compared to \(n\); if they are equal, control is transferred to statement \(S\) corresponding to that case, otherwise control is transferred to next case

3. If any of the cases is matched, control flow exits the case statement after executing statement \(S\), otherwise, the default statement is the last statement executed

Note that, in contrast to most programming languages (including C and Java), the control does not fall through to subsequent statements after matching one of the cases. Hence no break statements are needed. We also don’t allow duplicate cases, i.e. the values of \(n\) in each case must be unique, otherwise the compilation fails indicating a “duplicate case” error.

We extend the \text{label} function as follows

\[
\begin{align*}
\text{label}\left(\text{of } n: S\right) & = \text{of } [n]^{\ell}: \text{label}(S) \\
\text{label}(\text{default: } S) & = \text{default: } \text{label}(S) \\
\text{label}(\text{case } a \text{ cases default esac}) & = \text{case } a \text{ label}(cases) \text{ label}(default) \text{ esac}
\end{align*}
\]

where

\[ \ell = \text{newlabel()} \]
In order to update the flow function we introduce some notation first. Let \( \text{LCase} \) denote the set of labelled cases in a case statement and \( \mathcal{P}(\text{LCase}) \) its power set. We may then define two auxiliary functions:

\[
\text{next} : \text{LCase} \to \text{LCase} \\
\text{first} : \mathcal{P}(\text{LCase}) \to \text{LCase}
\]

where \( \text{next} \) returns the case that appears after the given case, while \( \text{first} \) returns the first case of a set of cases in a case statement.

We are now ready to extend the semantics:

\[
\begin{align*}
\text{init(} \text{of } [n]^\ell : S) & = \ell \\
\text{final(} \text{of } [n]^\ell : S) & = \text{final}(S) \\
\text{flow(} \text{of } [n]^\ell : S) & = \{(\ell, 1, \text{init}(S))\} \cup \text{flow}(S)
\end{align*}
\]

\[
\begin{align*}
\text{init(} \text{default: } S) & = \text{init}(S) \\
\text{final(} \text{default: } S) & = \text{final}(S) \\
\text{flow(} \text{default: } S) & = \text{flow}(S)
\end{align*}
\]

\[
\begin{align*}
\text{init(} \text{case a cases default esac) } & = \text{init(} \text{first(cases)} \text{) } \\
\text{final(} \text{case a cases default esac) } & = \text{final(} \text{default} \text{) } \cup \bigcup_{c \in \text{cases}} \text{final}(c) \\
\text{flow(} \text{case a cases default esac) } & = \bigcup_{c \in \text{cases}} \text{flow}(c) \cup \text{flow(} \text{default} \text{) } \cup \\
& \quad \cup \{(\text{init}(c), 1, \text{init(} \text{next}(c))\} \mid c \in \text{cases}\}
\end{align*}
\]

As before, we conclude with an example.

**Example 3.2.2.** The following, labelled pWhile program \( P \) makes use of a case statement.

\[
\begin{align*}
\text{var} \\
x & : \{1..3\}; \\
a & : \{0..10\}; \\
b & : \{0..10\}; \\
c & : \{-10..100\}; \\
\text{begin} \\
\text{case } x \\
\text{of } [1]^1 : [c:=a+b]^2 ; \\
\text{of } [2]^3 : [c:=a-b]^4 ; \\
\text{of } [3]^5 : [c:=a*b]^6 ; \\
\text{default: } [c:=0]^7 ; \\
\text{ esac; } \\
[\text{stop}]^8 ; \\
\text{end}
\end{align*}
\]
The flow of this program is given by:

\[
\text{flow}(P) = \{ \langle 1, 1, 2 \rangle, \langle 3, 1, 4 \rangle, \langle 5, 1, 6 \rangle, \langle 1, 1, 3 \rangle, \langle 3, 1, 5 \rangle, \langle 5, 1, 7 \rangle, \langle 2, 1, 8 \rangle, \\
\langle 4, 1, 8 \rangle, \langle 6, 1, 8 \rangle, \langle 7, 1, 8 \rangle, \langle 8, 1, 8 \rangle \}.
\]

### 3.2.4 Goto statement

Another language construct we introduce is a `goto` statement. It is not as widely used as the previous ones in everyday programming, but it has an interesting extension in the context of probabilistic programming. Before we expand on that, we introduce *tagged statement*, with syntax

\[
tag : S
\]

which lets us give identifiers to statements in our programs. Note that we use the word *tag*, and not normally used in that context *label*, to avoid confusion with labels which are part of our semantics. With a statement tagged in this manner, we may now transfer control to it from any point in the program using a `goto` statement, with the following syntax

\[
\text{goto} \ tag
\]

If none of the statements of the program has been tagged with *tag*, the compilation fails indicating “undefined tag” error. Since `pWhile` is a probabilistic programming language, we may add some probabilistic flavour to the `goto` statement. In particular, we might consider probabilistic `goto`, which transfers control to randomly chosen statement. Its syntax is as follows

\[
\text{goto} \ p_1 : \text{tag}_1 \text{ or } p_2 : \text{tag}_2 \text{ ro}
\]

where \( p_1 \) and \( p_2 \) are probabilities associated with transferring control to statements tagged \( \text{tag}_1 \) and \( \text{tag}_2 \) respectively. It transfers control to statement tagged \( \text{tag}_1 \) or \( \text{tag}_2 \) with (normalised) probabilities \( \frac{p_1}{p_1 + p_2} \) and \( \frac{p_2}{p_1 + p_2} \) respectively. As before, probability \( p_i \) is specified as a rational number using syntax

\[
r//s
\]

to represent \( \frac{r}{s} \).

We extend the `label` function as follows

\[
\begin{align*}
\text{label}(\text{tag} : S) &= \text{tag} : \text{label}(S) \\
\text{label}(\text{goto} \ tag) &= \text{[goto]}^\ell \text{ tag}
\end{align*}
\]

where

\[
\ell = \text{newlabel()}
\]

We have just introduced a new block, namely \([\text{goto}]^\ell\), and hence we must define the global operator corresponding to it. It is fairly straightforward in this
case, since its execution does not modify the state of variables. Therefore the local operator associated with goto is the identity operator \( \mathbf{I} \). We may hence extend Table 2.4 as follows

\[
T(\ell, \ell_k) = \mathbf{I} \otimes E(\ell, \ell_k) \quad \text{for } [\text{goto}]^\ell
\]

where \( \ell_k \) is assumed to be the label of block to which goto jumps. Trivially

\[
\begin{align*}
\text{init}(\text{tag} : S) &= \text{init}(S) \\
\text{final}(\text{tag} : S) &= \text{final}(S) \\
\text{flow}(\text{tag} : S) &= \text{flow}(S)
\end{align*}
\]

However, care has to be taken when defining the above functions for goto. We might be tempted to specify them in the following way:

\[
\begin{align*}
\text{init}([\text{goto}]^\ell \text{ tag}) &= \ell \\
\text{final}([\text{goto}]^\ell \text{ tag}) &= \{ \ell \} \\
\text{flow}([\text{goto}]^\ell \text{ tag}) &= \{ \}
\end{align*}
\]

The following example illustrates the fallacy of such approach.

**Example 3.2.3.** Consider the following program \( P \):

```plaintext
var
    x : {0..100};
    y : {0..100};
begin
[x:=0]^1;
tag: [x:=x+1]^2;
while [x<100]^3 do
    [goto]^4 \text{ tag};
[y:=y+1]^5;
od;
[stop]^6;
end
```

The flow of this program, computed using the function given above, is:

\[
\text{flow}(P) = \{ (1, 1, 2), (2, 1, 3), (3, 1, 4), (4, 1, 5), (5, 1, 3), (3, 1, 6) \}.
\]

However the triple \( (4, 1, 5) \) is clearly superfluous, since the goto statement will transfer the control flow to block 2. We would like to include triple \( (4, 1, 2) \) instead. Hence we need a way to treat goto as an exception and not transfer the flow to the statement appearing after it. The easiest way to do that is to amend the final and flow maps so that

\[
\begin{align*}
\text{final}([\text{goto}]^\ell \text{ tag}) &= \{ \} \\
\text{flow}([\text{goto}]^\ell \text{ tag}) &= \{ (\ell, 1, \text{block(tag)}) \}
\end{align*}
\]

39
where
\[ \text{block} : \text{Tag} \rightarrow \text{Lab} \]
maps tags to corresponding labels.

### 3.2.5 Constants

Finally, we introduce constants. Just as variables and arrays, any constant used in the program must be declared beforehand. In contrast to the above, constants must also be given a value when they are declared. One declares and initialises a constant \( c \) to value \( n \) using the following syntax

\[ c : n \]

placed in the declaration part of the program. One may then use \( c \) in any arithmetic or binary expressions, however one must not assign to a constant. This results in a compilation error.

### 3.2.6 Further extensions

The list of language extensions could be as long as we like. We designed and implemented (see Section 4.1) a few, but there are many more language constructs that could add value to the pWhile language. Among these are \texttt{break}, \texttt{continue}, single branch (i.e. no \texttt{else} part) \texttt{if} statements (relatively easy) or pointers, lists and function calls (harder).

### 3.3 Language choice

A hugely important part of any software project is the choice of implementation language. In our case, this task had even more significance, since three different languages are involved.

#### 3.3.1 pWhile

First of all, there is a probabilistic language which acts as an input language for our compiler. Since we are providing its implementation, the natural solution was to design it ourselves. That way, we could ensure that the language is compatible with Linear Operator Semantics and adapt its syntax to make the implementation easier. In fact, that task was done by H. Wiklicky, A. Di Pierro and C. Hankin [4] – what we present below is our take on what the motivation behind the design choices might have been.

As an example, consider the declaration part of every pWhile program. One could argue that it is archaic (Pascal is probably the most “modern” programming language having a very similar construct) and constitutes an unnecessary burden for the programmer. We prefer to view it as a minor overhead that makes
implementing \texttt{pwc} a lot easier. It allows our compiler to perform only one pass through the source code, thereby improving its performance. More importantly, it supplies the range of each variable, and hence determines the dimension of the state space. An alternative approach would involve the compiler performing an additional, initial analysis (i.e. a \textit{first pass}) of the code to pull out all the variables used in the program. This is an easy task and one that most compilers employ. The main difficulty lies in estimating the ranges of variables. One solution is to assign to each variable some default range. If it was too large however, the performance would suffer. If it was too small, most variables would end up being assigned values outside of their range. A better approach involves performing a more advanced analysis to estimate the range of each variable individually. That is a complex task in itself though and we decided that it is not worth the effort at that point. We believe that the impracticality of the declaration part only comes into play when the programs, and hence number of variables, get large. However, the state space of such programs is too big for our tool to handle (see Section 6.2). That limitation should therefore be addressed first.

Another important modification was the introduction of a \texttt{stop} statement. Recall that its meaning is an infinite loop - it transfers control to itself. The reason why we need it as part of \texttt{pWhile}'s syntax is to preserve the Markov chain properties. In particular, Markov chain never terminates - the sum of probabilities of transitions from each state must equal 1. However, we might obtain a behaviour very similar to termination by introducing an “isolated” state, from which the only possible transition is a loop to itself. An example is given below, where state 4 is an isolated state.

The \texttt{stop} statement and its semantics gives rise to such an isolated state (in fact, plenty of isolated states, since every unique configuration of variable values together with a \texttt{stop} block produce an isolated state). We may view reaching such a state as equivalent to termination.

The language is based on \texttt{While} - the most common choice when it comes to theoretical analysis of imperative programming language semantics. It is simple, easy to parse and can be extended without difficulty.
3.3.2 OCaml

The next important choice was the implementation language of our compiler. The following features of OCaml influenced our decision:

**Parsing and lexing facilities** We required our language of choice to offer a well-documented and fully-functional libraries for parsing and lexing. `ocamlyacc` and `ocamllex` closely follow the `Yacc` and `Lex` tools known from C and they enabled us not to reinvent the wheel.

**Imperative and functional features** OCaml is a functional language and therefore includes several extremely useful concepts such as: (i) higher-order functions, which, when used well, can significantly decrease the number of lines of code written, (ii) variant types, which enable us to easily represent different kinds of statements and expressions, (iii) pattern matching, which is especially useful when developing a compiler to match against the variant types mentioned above (in contrast to using a *visitor pattern*, which is a solution typically used in Java for instance). However, in contrast to, most notably, Haskell, OCaml combines its functional flavour with imperative features, simplifying I/O operations, introducing mutable values or hash tables. Purists might regard it as a weakness, we appreciate the flexibility it offers.

**Static type checking and type inference** OCaml is *statically typed*, which means that the compatibility between actual and formal types is checked at compilation time. This helps to avoid bugs caused by typographical errors and contributes to execution safety. However, no type information needs to be specified within the code. That is possible thanks to *type inference*, which means that the language alone can deduce the most general type of expressions and declarations therein. Although it might sometimes lead to code being hard to understand (see Section 4.2.1 for how we address this), it is quite convenient at the time of code development.

With hindsight, we are satisfied with our choice. The code produced is concise and writing it was a pleasure. Moreover, due to heavy use of higher-order functions and the static type checking our code had virtually no bugs (in other words, “If it compiles, it works”).

3.3.3 Julia

Finally, there is the language which does the “heavy work”, i.e. computes the Markov chain generator corresponding to the compiled probabilistic program. Its main requirement was to be well-suited to numerical computations, in particular matrix manipulation. The most popular such environment is certainly MATLAB. Unfortunately, it is a proprietary software and we aimed to avoid using such tools. Therefore we had to consider a free alternative to MATLAB.
Several such languages exist, with the most notable ones being GNU Octave, Scilab, Sage or Julia. The first of these, GNU Octave, is arguably the most popular one and was used in the reference tool. It was developed in the late 1980s and therefore it comes with a great support and documentation. It is mostly compatible with MATLAB and is generally regarded as its main free alternative. Its only downside, albeit an extremely important one, is that it is slow (see below). Therefore our aim was to choose a language as fast as possible and see how that improves the performance of our tool. We decided to use Julia - a dynamic programming language, which is in very early stages of development (at the time of writing, the latest stable release was 0.4.5), but already offers great performance and many interesting features. As the creators of the language themselves describe it [2]: \textit{We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like MATLAB. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as MATLAB, as good at gluing programs together as the shell. (…) We want to write $A*B$ and launch a thousand computations on a thousand machines, calculating a vast matrix product together.}

To support one of those claims, Julia was compared to a selection of other languages on a set of benchmarks. We present some of the results below:

<table>
<thead>
<tr>
<th></th>
<th>Julia</th>
<th>Python</th>
<th>MATLAB</th>
<th>Octave</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>2.11</td>
<td>77.76</td>
<td>26.89</td>
<td>9324.35</td>
<td>1.89</td>
</tr>
<tr>
<td>parse_int</td>
<td>1.45</td>
<td>17.02</td>
<td>802.52</td>
<td>9581.44</td>
<td>1.20</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.15</td>
<td>32.89</td>
<td>4.92</td>
<td>1866.01</td>
<td>1.29</td>
</tr>
<tr>
<td>mandel</td>
<td>0.79</td>
<td>15.32</td>
<td>7.58</td>
<td>451.81</td>
<td>1.11</td>
</tr>
<tr>
<td>pi_sum</td>
<td>1.00</td>
<td>21.99</td>
<td>1.00</td>
<td>299.31</td>
<td>1.00</td>
</tr>
<tr>
<td>rand_mat_stat</td>
<td>1.66</td>
<td>17.93</td>
<td>14.52</td>
<td>30.93</td>
<td>2.96</td>
</tr>
<tr>
<td>rand_mat_mul</td>
<td>1.02</td>
<td>1.14</td>
<td>1.12</td>
<td>1.12</td>
<td>1.42</td>
</tr>
</tbody>
</table>

Table 3.1: Benchmark times relative to C (smaller is better) [1]

So Julia is almost as fast as C, but has built-in functions for matrix manipulation and was designed with high-performance numerical computing in mind. The main downside of Julia is its young age, which means that the community is still quite small and there is not as much support online as for other languages. One usually has to refer to the official Julia manual when looking for help. Fortunately, it is well-written and answers most questions.
Chapter 4

Implementation

With the preliminary theoretical work completed and a good grasp of Linear Operator Semantics, we may now move on to the main part of the project - implementation. Conceptually, it could be divided into three parts: (i) implementing the language extensions described in Section 3.2, (ii) refactoring/reimplementing the existing tool, (iii) developing the Julia back end from scratch. We now describe those tasks in some detail.

4.1 Language extensions

The first part was the most straightforward one. Most of the hard work has already been done on paper, what remained was to transform those ideas into code. Because the tool design closely follows LOS, most of the changes amounted to extending:

- Lexer and parser to support new syntax
- Statement and Label modules with new (labelled) statements
- Flow module and its functions init, final and flow
- Block module to recover blocks from new statements.

Additionally, we had to implement a mapping from tags to labels to handle goto statement. We deliberated extensively whether to use a hash table or an association list for that task, and eventually chose the latter, since it required less code to be written and seemed more elegant. Moreover, due to small size of pWhile programs, the lookup overhead is negligible.

The more difficult part was to add support for arrays and constants. As a preliminary action on a way to achieve that, we introduced a symbol table holding meta data of variables (their range), arrays (their size and elements’ range) and constants (their value). We defer further discussion until Section 4.4 which describes the back end of our tool.
The correctness of introduced language extensions was verified by developing \texttt{pWhile} programs using the new constructs, compiling them with \texttt{pwc} and checking that the execution proceeds as expected.

4.2 Code refactoring

One of the main goals of the project was to refactor (where necessary, reimplement) the existing OCaml code, developed by the supervisor of this project, Dr Herbert Wiklicky. Being written over many years, it had many inconsistencies, repetitions and redundancies. Below we present our attempt to eliminate those deficiencies, by classifying the refactorings into several categories and describing each of them in some detail. Furthermore, to make it more precise, a few code examples are given in Appendix A.3.

4.2.1 General

We start by giving a rundown of general changes made to the code to ensure it follows best coding standards.

\textbf{Comments} We have added comments to the code. In general, we followed a principle that well-written code requires very few comments. However, as mentioned above, OCaml has type inference, which means that type annotations are not required in the code. Therefore, it might sometimes be unclear what are the types of arguments passed to certain functions. In such cases, we include this information as comments. We also often add a concise description of what a given function does, especially for the more complex ones. This is rarely the case for the OCaml code, which is mostly self-explanatory, but comes into play for Julia backend, where the majority of matrix manipulations are far from trivial. In writing the comments, we followed the formatting guidelines specified in the manuals. In particular, the documentation of Julia functions can be accessed at the REPL and the documentation of OCaml files can be auto-generated in various formats using \texttt{ocamldoc} tool.

\textbf{Extracting common functionality} This point should be self-explanatory. Avoiding repetition is one of the most fundamental principles of software engineering. It was especially applicable to parts of the tool responsible for generating Julia code. Similar patterns emerge repeatedly, such as code for creating and calling functions, assigning to variables, or initialising and populating a dictionary. We therefore introduced plenty of auxiliary functions for the above tasks. As a result, the amount of code decreased and, more importantly, became more flexible. Suppose for the sake of argument that the syntax for calling a function in Julia changed. Previously, we would have had to make plenty of changes in the code to adapt it to the
modification. Currently it would require a single change to the auxiliary
function responsible for generating Julia code corresponding to a function
call. In the extreme case, one might want to replace Julia with another
language. Current implementation allows such a transition to happen rela-
tively effortlessly.

**Rewriting existing functions** We identified several functions whose imple-
mentation was suboptimal. In most cases, they were unnecessarily com-
plicated. See Appendix A.3 for an example.

**More descriptive names** In order to make the code more readable, we modi-
fied selected function and variable names to make them more meaningful.
Again, we attempted to find the right balance between expressiveness and
conciseness of names used.

### 4.2.2 Handling output

Probably the biggest change in the code corresponds to the way output is man-
aged. There are currently two kinds of outputs that `pwc` produces. Firstly, there
is standard output, i.e. whatever the tool prints to the console during the com-
pile. By default, it consists of the variables declared in the program along
with their ranges, the blocks of the program and the control flow. Secondly,
there is output to the Julia file which constitutes the result of the compilation.
The existing code treated those two cases separately, thereby introducing plenty
of repetition, since the two outputs have a lot in common. To address this, we
introduced the concept of “default printer” and “special printers”. In particular,
every statement and expression have a default string representation, provided by
the default printer. However, if one requires an alternative way of printing any
statement or expression, they can specify it by defining an appropriate special
printer. For instance, we define a `julia_printer` which overrides the default
printer with Julia-specific ways of printing expressions. A code example can be
found in Appendix A.3.

### 4.2.3 Representing weights

Another substantial change we introduced is the way in which weights are repre-
sented. Recall that each flow triple is of the form \((d, w, s)\), where \(d, s\) are blocks
and \(w\) is the weight, i.e. a probability of transition from block \(d\) to block \(s\). For
the deterministic statements \(w = 1\), however when \(d\) is a `choose` block then \(w\)
is a rational number between 0 and 1. In the existing code, weight was rep-
resented by a variant type with two alternatives - `CWeight`, holding an integer,
and `PWeight`, holding a string. That approach had some weaknesses however:
(i) pattern matching had to be employed every time weight was handled in the code
to match against two different variants, resulting in unnecessary complexity, (ii)
the parser accepted any string to represent weight in a `choose` statement, in
particular non-numerical strings, (iii) the string had to be parsed (which could fail by (ii)) to recover the underlying probability.

The new approach addresses all those issues. Weight is now a type with single variant and holds two integers, i.e. the numerator and denominator of the rational number representing the probability. The parser now expects the weight to either be an integer or a rational number and indicates a parsing error when the value does not match this pattern. This is clearly preferable to having to deal with invalid input explicitly.

4.2.4 Miscellaneous

We have also identified and fixed a couple of minor flaws in the reference tool which do not deserve a separate paragraph but are worth mentioning:

- The type used to represent the LHS of an assignment (as part of the syntax tree of the program) in the reference tool is aexpr (i.e. arithmetic expression), which is arguably too general (e.g. it does not make sense to assign to x+y). We have therefore introduced a variant type varref, which stands for “variable reference”, supposed to accurately represent expressions which can be assigned to. As a result, our code is slightly cleaner as one can see in Appendix A.3.6

- The reference tool does not allow binary expressions to be on the RHS of an assignment. Since bool is a valid type in pWhile, one might expect such a feature to exist and therefore we implemented it.

4.3 Improving the functionality and reliability

One of the main weaknesses of the existing tool was its lack of user friendliness. Since our aim was to produce software that could be shared with and used by other people, we had to address this issue.

4.3.1 Lexer modifications

The main usability flaw we identified was the ungraceful way in which pwc crashed in presence of some syntactically wrong input programs. On reaching incorrect syntax, the parser would raise an exception, which would then be passed on to the user, who would receive no indication as to which part of the program was invalid. The failures were caused by programs which used sequences of characters that were not expected by the lexer. Recall that lexer transforms a program into a sequence of tokens, which are understood by the parser. To “configure” the lexer, one has to specify rules indicating what patterns correspond to what tokens. For example, “:” could be associated with token COLON and “+” with PLUS. There are also some complex rules, which use regular expressions to define string patterns, such as variable names. However, if a certain sequence
of characters in a given program is not matched by any of the rules, the lexer fails, raising an exception. If it is not caught, the compiler crashes and the user is presented with an incomprehensible message. It could be prevented in two ways: (i) either introduce a rule that would match any sequence of characters (a “catch all” rule), or (ii) catch and handle the exception thrown by the lexer. The solution we used does in fact both of the above. We defined the extra rule as the last one, which ensures that it is matched only if all the others are not. When matched, it raises an UnexpectedToken exception which we catch in the main program and print a message indicating the position of the unexpected character.

4.3.2 Semantic analysis

One of the features which we extended the existing tool with is semantic analysis of the supplied program. For the most part, it involves verifying that variable references are used correctly. In particular, following properties are checked:

- each reference must be valid, i.e. refer to a variable declared in the declaration part of the program
- arrays must be referenced in array context, i.e. using \texttt{a[i]} syntax
- constants must not be assigned to
- all tags must be matched, i.e. any tag used as part of a \texttt{goto} statement must also be a part of some tagged statement

Failure of a program to satisfy any of the above results in a compilation error. The properties are verified in a recursive manner by walking the syntax tree of the program and checking each statement on the way. To give a flavour of work completed, we give a code example in Appendix A.2.

4.3.3 Addressing other minor deficiencies

We conclude this section by listing a few minor flaws of the existing tool, which we identified and fixed or small extensions which we have implemented:

- the \texttt{Makefile} now reflects all the dependencies between the submodules of \texttt{pwc} and compiles the program in the first attempt (before, one had to run \texttt{make} command twice, because the first execution usually resulted in a compilation error)
- the tool no longer crashes when it is invoked without any arguments from the command line; instead, it prints a “usage message”
- \texttt{pwc} now accepts the full file name as its command line argument (e.g. \texttt{program.pw}); previously it required base name to be passed (e.g. \texttt{program}) - we believe the former is more intuitive
4.4 Julia back end

Any \texttt{pWhile} program is compiled into a Julia file, where the local and global operators introduced in Section 2.4 are defined. Executing it results in those operators being computed, which enables one to “run” the original \texttt{pWhile} program, by supplying appropriate initial state vector and post-multiplying it by the DTMC generator repeatedly. We now explain how the aforementioned Julia file is generated, by dividing the process into three parts to improve clarity of presentation.

4.4.1 Translating declarations

Recall from Section 2.3 that each variable in a \texttt{pWhile} program has a range and that we can arrange them according to the order in which they are declared, thereby associating an \textit{ordinal} to each. As we will see below, when defining the matrix operators it is essential to have this association of variables, ranges and ordinals at hand. It can be easily retrieved from the declaration part of the program and establishing it constitutes the first step of generating the Julia file. In particular, we introduce three dictionaries, which map: (i) variable names to their ranges, (ii) names to ordinals and (iii) ordinals to ranges. The process of generating Julia code that defines the above given the list of declarations is fairly straightforward. However, it might not be obvious how to handle arrays and constants. The latter are completely ignored - since they do not change value, there is no need to include them as part of probabilistic state. On the other hand, array of size $n$ is treated as $n$ separate variables, i.e. each of its elements is assigned a unique ordinal.

Apart from the dictionaries just described, we also pull out some extra information from the declaration part of the program, which will again be useful when defining the matrix operators. Firstly, we define an array that holds the dimensions of all variables. Secondly, we introduce a variable which stores the product of those dimensions, i.e. the dimension of all local operators.

4.4.2 Generating local operators

The most complex and most important part of the Julia file generation process is defining the local operators. Constructing all of them directly in the generated file would be highly impractical – hard to get right, extremely difficult to debug, and the resulting code would be incomprehensible. Therefore, we offload all the complexity related to computing the local operators to an auxiliary file (called \texttt{LOS.jl}). As a result, the generated Julia files are clear and readable for anyone who understands LOS, and the complicated functions responsible for generating matrices are defined separately and hence can be tested independently.

We now describe some implementation details of this part of Julia back end. We know from Section 2.4.4 that local operators arise from blocks. Hence to
generate all of them, one simply has to iterate through blocks of a given program and produce operator(s) corresponding to each of them. Generating the identity operator is trivial given its dimension (which we have by Section 4.4.1), for the other two the task is more involved and so we describe it below. The full code of the auxiliary file is presented in Appendix A.4 and it might be helpful to refer to it if our explanation feels unclear. We also present there several generated Julia files, which use functions defined in LOS.jl.

For each operator, we first consider the algorithm used in the function computing it in the auxiliary file and then outline the process of generating Julia code which calls the that function. Below, we sometimes refer to the generated Julia file as “compilation product”. Finally, note that we use some of the ideas present in the implementation of the reference tool, for which credit goes to Dr Herbert Wiklicky.

Filter Operator

We start with the easier of the two, the Filter Operator. The algorithm used to compute it follows very closely from the definition given in Table 2.2. It iterates over the variable states, and in each of them, the binary expression (i.e. the condition of the if of while statement) is evaluated. If it matches the expected truth value, the corresponding diagonal entry of the resulting matrix is set to one – otherwise it is set to zero. Consider now the parameters on which this algorithm operates. First of all, it needs the number of variable states, which determines the dimensions of the produced matrix. Next, it requires the binary expression \( b \) which gives rise to the operator. This is more complex, it is not even obvious at first what data structure to use to pass it to the function. However, we may observe that \( b \) itself is not necessary; all that is required is an “evaluation function” which computes the value of \( b \) in a given state.

We have not mentioned yet how do we represent state in the code. So far, we have seen two ways of doing it in general: one uses lexicographical ordering to identify each state with an integer, the other is a vector (tuple) listing indexes into each variable’s range. An example of both approaches is shown in Table 2.1. Note that the vector contains indexes into variables’ range rather than their actual values. We already remarked in Section 2.4.4 that such an abstraction is convenient from the point of view of defining matrix operators. It should not come as a surprise that we adopt this convention throughout the auxiliary file, in a sense that all the code in LOS.jl assumes that each variable \( v \) has values \( 1, 2, \ldots, n \), where \( n \) is the dimension of \( v \). To avoid repeating ourselves over and over again, every time we refer to variable values in the context of the helper file, we allude to the indexes into their range. We also call the vector specifying values of variables a “variable state vector”.

Going back to our algorithm, the integer representation of state is clearly preferable from the point of view of iterating over it. However, the implementation of the evaluation function is more straightforward when state is passed as
a vector of values. Since our aim is to keep the compilation product simple and offload all the complexity to the auxiliary file, we need a way to transform one representation into the other (so that we can iterate using integer representation but pass variable state vector to the evaluation function). In order to do that, we use the following correspondence: If there are $n$ ordered variables of dimensions $d_1, d_2, \ldots, d_n$ (and hence $d_1 \cdot d_2 \cdot \ldots \cdot d_n$ states) then in state $k$, the $i^{th}$ variable takes its $(1 + ((k - 1) \ div \ d) \ mod \ d_i)^{th}$ value where

$$d = \begin{cases} d_{i+1} \cdot d_{i+2} \cdot \ldots \cdot d_n & \text{if } i < n \\ 1 & \text{otherwise.} \end{cases}$$

The formula can be verified on the example from Table 2.1. A similar correspondence can be devised mapping variable state vector to an integer representation. Both are implemented in the auxiliary file. Note that the formula uses dimensions of all variables, and hence they need to be passed to the algorithm. From the dimensions, one can easily recover the number of states, and so that parameter is no longer necessary.

We now have a full picture of how the function computing a filter operator proceeds. It takes three arguments: the dimensions of variables, the evaluation function and the expected value of the binary expression. It starts by initialising a sparse matrix full of zeros, of dimension equal to the number of variable states (which is a product of variable dimensions). It then iterates over those states, using their integer representation given by lexicographical ordering. In each iteration, the variable state vector is recovered and supplied to the evaluation function which returns the value of the binary expression. If this value matches the expectation, the appropriate diagonal entry (explicitly, $i^{th}$ entry in the $i^{th}$ iteration) of the matrix is set to 1.

We now outline how the code calling the just described function is generated. Among the three arguments of this function, two are readily available: the array with variable dimensions was defined before and the expected value of the binary expression is either true or false, depending on the branch of the conditional. The only work that has to be done involves defining the evaluation function. We recall that it must take one argument, an array of indexes into variables range, and return the value of the binary expression $b$. Therefore, all we need to do is translate $b$ into valid Julia syntax, substituting variable names by their values. The translation proceeds in a recursive manner by walking the syntax tree corresponding to $b$ and essentially reduces to converting pWhile arithmetic symbols into their Julia counterparts, most of which are the same (One exception is integer division, represented by “/” in pWhile and “\text{div}” in Julia). Substitution uses the array passed as argument and dictionaries mentioned in Section 4.4.1. In particular, the following operations are applied to each variable name to recover its value in a given state:

1. the ordinal of the variable is obtained from one of the dictionaries
2. using the returned ordinal, the index into variable’s range is retrieved from the variable state vector
3. given the index, the value of the variable is restored from its range, which is obtained using the dictionary mapping variable names to their ranges

**Update Operator**

We now outline how update operator is computed. We start by recalling its two types: firstly, there is a constant update operator, in which the assigned value is a constant, and secondly, the more general update operator where the RHS of an assignment is an arbitrary expression. The generation of the first one is an easy task and uses Kronecker product repeatedly to compute the resulting matrix. The process of producing the more general update operator resembles closely what we described above for filter operator. We again iterate over the variable states and in each of them invoke the supplied evaluation function. In this case, it computes the value of an arithmetic, rather than binary expression. The main difference is how we handle the result. Previously, only two values were possible: *true* or *false*, and we simply compared them with the given, expected value. Now it is more complicated – the value is an arbitrary integer. We recall our convention of treating variable values as indices into their range in the context of the auxiliary file. We may therefore enforce the value returned from the evaluation function to be an index into range of the variable assigned to. Initially, we compute the update operator in Section 2.4.4, it should now be clear what to do for each state: We first convert the integer representation into the variable state vector representation. Then, we invoke the evaluation function to retrieve the value of the expression in the current state and use that new value to update the state vector. This action requires the ordinal of the variable assigned to, which indicates which entry of the variable state vector should be modified. Hence such an ordinal is a parameter of the algorithm and an argument for the associated function in \texttt{LOS.jl}. Finally, converting the updated state vector to its integer representation gives us the index into the resulting matrix which we must set to 1. Explicitly, in the $i^{th}$ iteration, we are assuming that the state before executing the assignment is the $i^{th}$ state. If executing the assignment puts us in state $j$, then the $ij$ entry of the update operator is set to 1.

For completion, we also sketch how the update operator is computed when the LHS of an assignment is an entry of an array. The algorithm is similar to the one presented for variable assignment, albeit slightly more complex. In particular, it has two more parameters, the first one being the size of the array (which can be retrieved from the symbol table), and the second one another evaluation function, which computes the position in the array. In each state, the algorithm computes the index of the array entry which is being assigned to (using the extra evaluation function), and if it is valid, updates the variable state vector with the computed (using the usual evaluation function) value of an expression. Finally, it sets the appropriate entry of the resulting matrix to 1. If the computed array position is invalid, the algorithm proceeds to the next state,
which results in an all zero row in the corresponding local operator.

### 4.4.3 Generating global operators

Generating Julia code which defines the global operators is very easy once the local operators are in place. It involves iterating over the flow triples and producing one operator for each of them exactly as described in Section 2.4.5. Finally, the DTMC generator associated with the program is the sum of all global operators.

### 4.4.4 Sparse matrices

We have seen how the dimension of local and global operators grows as the size of the program increases. It quickly becomes clear that memory limitations imposed by the hardware on which our programs run will impact their volume significantly. For example, a DTMC generator $G$ corresponding to a program with three variables, each with twenty values in its range, and ten blocks is a $80000 \times 80000$ matrix. An easy calculation shows that, assuming that its entries are 64-bits floating point values, $G$ occupies 51.2 GB which is more than any regular machine has. If we wanted to be more accurate and let the entries of $G$ be rational numbers (represented by two 64-bit integers) then the amount of required memory doubles.

Fortunately, we can diminish this restriction considerably by noting that the matrix operators are in vast majority of cases sparse which means that most of their entries are zero. This leads to a more efficient way of saving them in memory. The most straightforward format, called “dictionary of keys” (DOK), stores only nonzero entries as a mapping of $(\text{row}, \text{column})$ – pairs to values of elements. Alternative ways of handling sparse matrices include LIL, where each row has a corresponding list of $(\text{column}, \text{value})$ – pairs, COO, in which list of $(\text{row}, \text{column}, \text{value})$ – tuples is used, or the most efficient CSR and CSC formats, whose description one can find online [12]. The upshot is that the amount of memory required to store an $n \times n$ sparse matrix is in average case proportional to $n$, rather than $n^2$. For example, stored as a sparse matrix, the DTMC generator $G$ corresponding to the program mentioned above would require no more than 1 MB of memory. We have therefore alleviated the problem but we have not solved it completely. However, as we will see in Section 6.2, the main factor limiting the size of programs $\text{pwc}$ can handle efficiently is CPU speed rather than the amount of RAM.

### 4.5 Optimisation – tracking variables appearing in expressions

One of the shortcomings of the proposed semantics is that not all operators can be expressed as a Kronecker product, which leads to inefficient implementation
and reduces the size of programs which \texttt{pwc} can handle. This can be observed in Tables 2.2 and 2.3. In particular, \( U(x_k \leftarrow c) \) can be expressed as a Kronecker product, which allows for fast implementation, whereas \( U(x_k \leftarrow e) \) is expressed as a sum over all variable states, which makes the implementation inefficient. However, in some cases, Kronecker product can be used to compute the \( U(x_k \leftarrow e) \) operator. To illustrate this, we give the following example.

**Example 4.5.1.** Consider program \( P \):

```plaintext
var
x : {0..1};
y : {0..1};
z : {0..1};
begin
[x:=y]¹;
[x:=z]²;
[stop]³;
end
```

The update operator corresponding to the first assignment is:

\[
U(x \leftarrow y) = \begin{pmatrix}
1 & . & . & . & .
. & 1 & . & . & .
. & . & . & . & 1
. & . & . & . & .
1 & . & . & . & .
. & 1 & . & . & .
. & . & . & . & 1
. & . & . & . & .
. & 1 & . & . & .
\end{pmatrix}
\]

Clearly,

\[
U(x \leftarrow y) = \begin{pmatrix}
1 & . & . & .
. & 1 & . & .
1 & . & . & .
. & 1 & . & .
\end{pmatrix} \otimes \begin{pmatrix}
1 & 1
\end{pmatrix}
\]

where the \( 4 \times 4 \) matrix on the LHS of the Kronecker product can be regarded as \( U(x \leftarrow y) \) restricted to variables \( x \) and \( y \), and the \( 2 \times 2 \) identity matrix on the RHS of the Kronecker product is \( U(x \leftarrow y) \) restricted to \( z \). Note that even though value of \( y \) does not change as a result of the assignment, we cannot express the \( 4 \times 4 \) matrix as Kronecker product because value of \( x \) depends on the value of \( y \). Now
cannot be expressed as a Kronecker product of $2 \times 2$ or $4 \times 4$ matrices. The reason for that is that here first and third variables are involved in the assignment which makes it impossible to decouple them from the other variable, not involved in the assignment. However, both assignments have identical structure, which suggests that the update operators representing them are related in some way. Indeed, if we computed the matrix operator that “swaps” $y$ and $z$ (we make the notion of “swapping” more precise below), we could use it to express $U(x \leftarrow z)$ in terms of $U(x \leftarrow y)$, which we can compute efficiently using Kronecker product. To achieve that, we recall that the variable state vector may be represented as a Kronecker product of the distributions of individual variables. Hence, letting $v_x, v_y, v_z$ denote the distributions of $x, y, z$ respectively, we would like to find a matrix $K(1, 2)$ such that

$$(v_x \otimes v_y \otimes v_z) K(1, 2) = v_x \otimes v_z \otimes v_y$$

We will make a clever use of properties of Kronecker product to find $K(1, 2)$. First of all, in Section 2.2.3 we showed how to restore the commutativity of Kronecker product: for any matrices $A$ and $B$, we can find commutation matrices $P$ and $Q$ such that

$$P(A \otimes B)Q = B \otimes A$$

In the special case when $A$ and $B$ are row vectors, $P$ is a $1 \times 1$ identity matrix (this follows from the definition of commutation matrix, see Section 2.2.3), so we can drop it in the above equation. Going back to our example, we see that there exists a matrix $L$ such that

$$(v_y \otimes v_z) L = v_z \otimes v_y$$

What is left is to extend $L$ in a way that would preserve its action on $v_y \otimes v_z$ and act as identity on $v_x$. We saw a similar construction in Section 2.4.5 when we defined global operators. We again use the mixed-product property, which ensures that

$$(v_x \otimes (v_y \otimes v_z))(I \otimes L) = (v_x I) \otimes (v_y \otimes v_z)L$$

$$= v_x \otimes v_z \otimes v_y$$
Hence we can set \( K(1, 2) = I \otimes L \). We can compute \( L \) using the definition of the commutation matrix, which lets us calculate \( K(1, 2) \) as

\[
K(1, 2) = \begin{pmatrix}
1 & \ldots & \ldots & \ldots & \ldots \\
\ldots & 1 & \ldots & \ldots & \ldots \\
\ldots & \ldots & 1 & \ldots & \ldots \\
\ldots & \ldots & \ldots & 1 & \\
\ldots & \ldots & \ldots & \ldots & 1 \\
\end{pmatrix}
\]

As promised, we can now express \( U(x \leftarrow z) \) in terms of \( K(1, 2) \) and \( U(x \leftarrow y) \).

The idea is that applying operator \( U(x \leftarrow z) \) is equivalent to swapping \( y \) and \( z \), followed by applying \( U(x \leftarrow y) \), and finally swapping \( y \) and \( z \) back. Therefore

\[
U(x \leftarrow z) = K(1, 2)U(x \leftarrow y)K(1, 2)
\]

In the general case of a program with \( n \) variables we would like to find a swap operator \( K(i, j) \), which swaps \( i^{th} \) and \( j^{th} \) variables. This is more complicated since \( i \) and \( j \) need not be consecutive integers, which the above construction requires. One way to proceed would be to compute operators swapping variables \( i \) and \( i + 1 \), \( i + 1 \) and \( i + 2 \), ..., \( j - 1 \) and \( j \), and compute \( K(i, j) \) as a product of the above. This is inefficient however, and not particularly elegant. For a better solution we have to use the definition of the commutation matrix and make some clever observations. To keep the notation simple, we show how to compute the operator \( K(1, 3) \) which swaps \( x \) and \( z \) from the above program. Presented approach can easily be generalised.

First of all, we will find a matrix \( L \), such that

\[
(v_x \otimes v_z)L = v_z \otimes v_x
\]

This is the usual commutation matrix and it is defined in the following way

\[
L = \sum_{i=1}^{2} \sum_{j=1}^{2} (E_{ij} \otimes E_{ij}^T)
\]

**Example 4.5.2** (Commutation matrix). Suppose \( v_x = (\frac{1}{3}, \frac{2}{3}, 1) \) and \( v_z = (\frac{1}{4}, \frac{3}{4}) \), so

\[
v_x \otimes v_z = \begin{pmatrix} 1 \\ \frac{1}{12} \\ \frac{1}{4} \\ \frac{1}{6} \\ \frac{1}{2} \end{pmatrix}
v_z \otimes v_x = \begin{pmatrix} 1 \\ \frac{1}{12} \\ \frac{1}{4} \\ \frac{3}{4} \\ \frac{1}{2} \end{pmatrix}
\]

We therefore expect \( L \) to fix the first and last entries of \( v_x \otimes v_z \) and swap the other two.
Let

\[ L_1 = E_{11} \otimes E_{11}^T = \begin{pmatrix}
  1 & \ldots & \\
  \vdots & \ddots & \\
  \vdots & & 1 \\
\end{pmatrix} \]

\[ L_2 = E_{12} \otimes E_{12}^T = \begin{pmatrix}
  \vdots & \ldots & \\
  \vdots & \ddots & \\
  \vdots & & 1 \\
\end{pmatrix} \]

\[ L_3 = E_{21} \otimes E_{21}^T = \begin{pmatrix}
  \vdots & \ldots & \\
  \vdots & \ddots & \\
  1 & \ldots & \\
\end{pmatrix} \]

\[ L_4 = E_{22} \otimes E_{22}^T = \begin{pmatrix}
  \vdots & \ldots & \\
  \vdots & \ddots & \\
  \vdots & & 1 \\
\end{pmatrix} \]

so that

\[ L = L_1 + L_2 + L_3 + L_4 \]

Then

\[
(v_x \otimes v_z) L = (v_x \otimes v_z)(L_1 + L_2 + L_3 + L_4) \\
= (v_x \otimes v_z)L_1 + (v_x \otimes v_z)L_2 + (v_x \otimes v_z)L_3 + (v_x \otimes v_z)L_4 \\
= (v_x \otimes v_z)(E_{11} \otimes E_{11}^T) + (v_x \otimes v_z)(E_{12} \otimes E_{12}^T) \\
+ (v_x \otimes v_z)(E_{21} \otimes E_{21}^T) + (v_x \otimes v_z)(E_{22} \otimes E_{22}^T) \\
= (v_x E_{11} \otimes v_z E_{11}^T) + (v_x E_{12} \otimes v_z E_{12}^T) \\
+ (v_x E_{21} \otimes v_z E_{21}^T) + (v_x E_{22} \otimes v_z E_{22}^T) \\
= \left( \frac{1}{12}, 0, 0, 0 \right) + \left( 0, 0, \frac{1}{4}, 0 \right) + \left( 0, \frac{1}{6}, 0, 0 \right) + \left( 0, 0, 0, \frac{1}{2} \right) \\
= \left( \frac{1}{12}, \frac{1}{6}, \frac{1}{4}, \frac{1}{4} \right) \\
= v_z \otimes v_x
\]

So \( L \) is a sum of four components \( L_1, \ldots, L_4 \), where \( L_i \) extracts the \( i \)th entry of \( v_x \otimes v_z \) and places it in the appropriate position.

Suppose now that \( v_y = (\frac{1}{2}, \frac{1}{2}) \). Then

\[
(v_x \otimes v_y \otimes v_z) = \left( \frac{1}{24}, \frac{1}{8}, \frac{1}{12}, \frac{1}{8}, \frac{1}{12}, \frac{1}{4}, \frac{1}{12}, \frac{1}{4} \right) \\
(v_z \otimes v_y \otimes v_x) = \left( \frac{1}{24}, \frac{1}{12}, \frac{1}{24}, \frac{1}{12}, \frac{1}{8}, \frac{1}{4}, \frac{1}{8}, \frac{1}{4} \right)
\]

Hence we might imagine operator \( K(1,3) \) as consisting of four components \( K_1, \ldots, K_4 \), each of which extracts two entries of \( v_x \otimes v_y \otimes v_z \) and places them
in appropriate positions. Each $K_i$ should be closely related to $L_i$. In fact, we may define

$$K(1, 3) = \sum_{i=1}^{2} \sum_{j=1}^{2} (E_{ij} \otimes I \otimes E_{ij}^T)$$

Intuitively, the additional identity matrix reflects the fact that $v_y$ stays in its position. We may now check that indeed

$$ (v_x \otimes v_y \otimes v_z)K(1, 3) = (v_x \otimes v_y \otimes v_z)(E_{11} \otimes I \otimes E_{11}^T) + (v_x \otimes v_y \otimes v_z)(E_{12} \otimes I \otimes E_{12}^T) + (v_x \otimes v_y \otimes v_z)(E_{21} \otimes I \otimes E_{21}^T) + (v_x \otimes v_y \otimes v_z)(E_{22} \otimes I \otimes E_{22}^T)$$

$$= v_x E_{11} \otimes v_y I \otimes v_z E_{11}^T + v_x E_{12} \otimes v_y I \otimes v_z E_{12}^T + v_x E_{21} \otimes v_y I \otimes v_z E_{21}^T + v_x E_{22} \otimes v_y I \otimes v_z E_{22}^T$$

$$= (\frac{1}{24}, 0, \frac{1}{24}, 0, 0, 0, 0, 0) + (0, 0, 0, 0, \frac{1}{8}, 0, \frac{1}{8}, 0) + (0, \frac{1}{12}, 0, \frac{1}{12}, 0, 0, 0, 0) + (0, 0, 0, 0, 0, \frac{1}{4}, 0, \frac{1}{4})$$

$$= (\frac{1}{24}, \frac{1}{12}, \frac{1}{24}, \frac{1}{12}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8})$$

$$= v_z \otimes v_y \otimes v_x$$

**Implementation**

We now briefly outline how the above idea is implemented, however due to its high complexity, we will not go into much detail. Readers particularly interested in it are referred to Appendix A.4. First of all, with every expression $e$ in a given program, we associate a set of variables which appear in $e$. In the auxiliary file, we define new functions computing the update and filter operators. Their signature is similar to the regular functions described in Section 4.4.2, however they take an extra argument, which is the list of ordinals of variables appearing in the expression $e$ with which the operator is associated. The algorithm used in those new functions proceeds as follows:

1. It determines what variable swaps must take place so that variables appearing in $e$ have the lowest ordinals.

2. It computes the new array of variable dimensions, containing only dimensions of variables which appear in $e$, in the right order.
3. It calculates the local operator $R$ corresponding to the restricted set of variables using the regular technique of iterating over states and retrieving the value of an expression in each of them.

4. It extends $R$ by forming its Kronecker product with local identity operators acting on variables not appearing in $e$

5. Finally, it computes the swap operator $K$, multiplies it by $R$ and by itself, and returns the result.

By default, \texttt{pwc} does not employ the described optimisations. However, one can enable them using command line options. In Section 6.2.2 we present how do they affect the performance of the tool.
Chapter 5

Testing framework

To test our software, in addition to developing a suite of unit tests, we created a testing framework, specifically targeted at verifying the correctness of \texttt{pwc}.

5.1 Motivation

The best way to test a compiler is to supply it with plenty of examples and verify the produced output. However, in case of \texttt{pwc} this task is not as easy as it might seem. We must keep in mind that our compiler produces a file, which defines the set of matrices associated with the program, most important of which is the DTMC generator. For bigger programs, its dimensions are large and it is unfeasible to compute it manually to compare the expected vs actual. What can be done is applying this generator repeatedly to some initial state vector and verifying that the transitions are as expected. However, this in itself is not trivial either. One must first compute the initial state vector by multiplying (using Kronecker product) the distributions of individual variables and the block vector. One then multiplies the state vector by the generator matrix and obtains an updated state vector. This alone is in general not too meaningful either, since the result is just a (possibly) large vector with a few nonzero entries. To verify that it matches expectations, one must compute the expected state vector by, again, multiplying together the expected distributions of individual variables and the expected block vector. As the size of the program grows, the dimensions of those vectors increase and there are more transitions until the program “terminates” (more formally, the final \texttt{stop} statement is reached). Hence the manual testing of \texttt{pwc} is an extremely time-consuming and arduous process. To save ourselves time and effort, we automated it.

5.2 Overview

The flowchart below depicts the main steps of test generation. To test a given \texttt{pWhile} program $P$, one must create a file specifying expected executions of $P$. 

Each such file must be named after the program it tests, with extension .t. Its syntax is described in detail in Section 5.3. In a nutshell, it defines one or more expected executions of the program by giving the initial state vector and the state vector after any number of execution steps. The runtests.py script runs the tests, which involves the following for each .t file:

1. Parse it to recover the expected executions.
2. Generate a Julia test file with all the test cases.
3. Execute the generated Julia test.
4. Print the outcome

![Diagram of test process]

### 5.3 Test files syntax

The test file is made up from one or more test cases:

\[
\begin{align*}
\text{<test file>} &::= \text{<test cases>} \\
\text{<test cases>} &::= \text{<test case>} \\
&\quad| \quad\text{<test case>} \text{<test cases>}
\end{align*}
\]

Each test case specifies one expected execution of a tested program and has the following structure:

\[
\text{<test case>} ::= \text{begin} \text{<body>} \text{end}
\]

where body of the test case is of the form:
\[
\text{let } \text{body} ::= \text{init state} \text{ step states}
\]

and consists of an \textit{initial state}, specifying variable values before the execution of the program, followed by one or more \textit{step states}, specifying variable values after any number of executed steps. Formally:

\[
\begin{align*}
\text{init state} & ::= \text{init} : \text{state} \\
\text{step state} & ::= \text{step } n : \text{state} \\
\text{step states} & ::= \text{step state} \\
& \quad | \quad \text{step state} \text{ step states}
\end{align*}
\]

where \text{state} specifies values of variables and the current block. State can be \textit{deterministic}, in which case it simply lists values of variables, or \textit{probabilistic}, which means that different states are possible with certain probabilities associated to each. The only catch is that deterministic state is not entirely deterministic – it allows value of a variable to be specified as a distribution, thereby providing a shorthand syntax for defining probabilistic state. Formally:

\[
\begin{align*}
\text{prob values} & ::= r/s : n \\
& \quad | \quad r/s : n \text{ or prob values} \\
\text{values dist} & ::= n \\
& \quad | \quad v = \text{values dist} \\
\text{var state} & ::= \text{var state} \text{ ; block } = n \\
\text{det state} & ::= \text{var state} \text{ ; block } = n \\
\text{prob state} & ::= r/s : \text{det state} \\
& \quad | \quad r/s : \text{det state} \text{ prob state} \\
\text{state} & ::= \text{det state} \\
& \quad | \quad \text{prob state}
\end{align*}
\]

The following example should make it clear.

\textbf{Example 5.3.1.} Consider program \(P\):

\[
\begin{align*}
\text{var} & \\
x & : \{0,1\}; \\
y & : \{0,1\}; \\
z & : \{0,1\}; \\
\text{begin} & \\
[x?={0,1}]^1; \\
y := x^2; \\
[\text{choose}]^3 1/3: [z:=0]^4 \text{ or } 2/3: [z:=1]^5 \text{ ro}; \\
[\text{stop}]^6; \\
\text{end}
\end{align*}
\]

The following test case corresponds to an expected execution of the program, where we assume that initially all variables are zero.
begin
init: x=0, y=0, z=0; block=1
step 1: x=1/2:0 or 1/2:1, y=0, z=0; block=2
    1/2: x=1, y=1, z=0; block=3
step 2: 1/2: x=0, y=0, z=0; block=3
    1/2: x=1, y=1, z=0; block=3
step 3: 1/6: x=0, y=0, z=0; block=4
    1/6: x=1, y=1, z=0; block=4
    2/6: x=0, y=0, z=0; block=5
    2/6: x=1, y=1, z=0; block=5
step 4: 1/6: x=0, y=0, z=0; block=6
    1/6: x=1, y=1, z=0; block=6
    2/6: x=0, y=0, z=1; block=6
    2/6: x=1, y=1, z=1; block=6
end

Note that variables in the test case are being set in the same order in which they appear in the program declaration - this is required for the tests to work correctly. Note also that we could have written the distribution after step 1 as:

    step 1: 1/2: x=0, y=0, z=0; block=2
            1/2: x=1, y=0, z=0; block=2

For convenience, we used an alternative, shorter syntax. Finally, note that since the first block in the program is always the first one to be executed, the initial state will always contain block=1, which is therefore redundant. It is required to include it however, for the sake of consistency with other states.

5.4 Implementation

The testing framework is implemented in Python, which is a natural choice when it comes to small programs such as this one. It enabled us to implement it quickly and efficiently, thanks to its clean, intuitive syntax and plenty of built-in data types and modules. We also used PLY, which is a Python implementation of Lex and Yacc, to avoid parsing the test files ourselves.
Chapter 6
Evaluation

This chapter describes efforts undertaken to evaluate \texttt{pwc}. It was not our aim to produce a tool with extraordinary user experience – as any other compiler, \texttt{pwc} has a very basic functionality and is operated from the command line. We have improved this aspect of the tool as presented in Section 4.3. In any case, we believe that the defining factor of our implementation is its performance. Therefore, we focus solely on the quantitative evaluation of the compiler.

6.1 Correctness

Our priority was to ensure the correctness of our tool. To this end, we developed the testing framework described in Section 5. At the time of submitting the report, we had eighteen example \texttt{pWhile} programs, along with test files specifying the expected executions of each of them. The tests are exhaustive in terms of variety of language constructs used and different patterns of execution. Moreover, we developed nine example programs which are syntactically or semantically incorrect, to ensure the tool successfully identifies those errors. It goes without saying that all the tests were passing. Furthermore, for selected programs we manually compared the matrices produced by the reference tool, by our tool used without optimisations, and by our tool with optimisations enabled. In all cases, the operators were the same.

6.2 Performance

The main reason for developing the completely new back end of \texttt{pwc} in Julia was to improve the performance of the tool. Before describing how to measure the efficiency of our compiler, we specify what exactly do we want to measure. Recall that each \texttt{pWhile} program $P$ is compiled into a file which defines all the matrix operators associated with $P$. In our case, this is a Julia file, in the reference tool, it was an Octave file. We could easily imagine it being implemented in MATLAB, C or Python. Importantly, computing all those operators is the bottleneck of
our tool. As the program size grows, the number of states becomes bigger and therefore the matrix operators associated with it become larger. If a program contains many blocks, including complex assignments, the computation of those matrices becomes non-trivial and time consuming. Therefore, the time taken to compute the DTMC generator (which involves computing all the local and global operators) corresponding to a given program acts as a good indication of the performance of \texttt{pwc}. We might compare this value for different implementations of \texttt{pwc}'s back end to see which one is most efficient. Crucially, it may enable us to see whether Julia back end outperforms the one implemented in Octave. It may also be helpful to see how this performance changes when the number of states increases for the same program (which can be achieved by increasing ranges of variables). Could we observe linear, polynomial or exponential relation between the number of states and the execution time?

In order to answer those questions, we created a little framework, which enables one to define “test environment” and run the performance tests automatically.

### 6.2.1 Performance testing framework

The main advantage of our framework is that it enables one to easily test the performance of the compiler on a given program with varying ranges of variables. To achieve that, we introduce two new file types, both of which are very closely related to \texttt{pWhile} programs. First of them has extension \texttt{pws}, which stands for “pWhile stub”. It is any \texttt{pWhile} program with the declaration part removed. The second one is a .\texttt{ds} (which stands for “declarations”) file - it consists of one or more declaration parts, each of them separated by keyword \texttt{or}. Examples can be found in Section 6.2.2, where we present benchmarks we developed. Those files are used in the following way: one defines the program on which they want to test \texttt{pwc}'s performance by creating a stub file. One can then specify as many different declaration parts corresponding to this program as they wish in the .\texttt{ds} file. The requirement is that both files have the same base name. The framework would then pick up both files, join each declaration part with the main program and compile it using the predefined implementations of the compiler (in our case, it used the Octave and Julia implementations). It would then measure the time taken to execute the generated files defining the LOS semantics of the program and generate a text file reporting all the results. One can also specify a regular \texttt{pWhile} program, which would be tested in a similar fashion. An extract from the “report file” can be found in Appendix A.5. Moreover, to achieve more genuine results, one can specify the number of times the generated files are executed. The reported time is then the average of all runs.

The script that does all of the above is implemented in Python, for similar reasons as the ones outlined in case of the testing framework.
6.2.2 Benchmarks and results

We now briefly describe the programs which we used to evaluate the performance of our compiler and present the results. Each program was compiled using the Octave implementation, Julia implementation and Julia implementation with optimisations enabled (which we refer to as “optimised Julia implementation”).

The tests were run on a standard DoC\textsuperscript{1} lab machine, with a i7-3770 processor running at 3.4GHz and 15GB of RAM.

The first program (\texttt{empty.pws}) is a very simple one, consisting of a single \texttt{stop} statement, declaring three variables with different ranges defined in the corresponding \texttt{empty.ds} file. Both files can be found in Appendix A.5.1. Its purpose is to measure the overhead associated with each implementation, i.e. how long does it take to set up the data structures described in Section 4.4. Since the only operator computed is the identity operator, we might also learn something about the performance of languages used to implement respective back ends. The results of running performance tests on this file are depicted in Figure 6.1. We do not include the results for optimised Julia implementation in the plot to improve clarity – they were almost identical to those displayed for regular implementation. The results suggest that the overhead associated with Julia implementation is bigger, which is the result of initialising the dictionaries holding the variable mappings (the Octave implementation does not initialise any data structures). However, the language itself is faster than Octave, as shown by the compilation time corresponding to $10^8$ states, where Julia outperforms Octave, regardless of bigger set up overhead. Finally, when the number of states was $10^9$, both implementations failed to compute the operators due to insufficient memory.

Another test file, called \texttt{population.pws} performs some real computation. In fact this program models the probabilistic riddle introduced in Example 1.1.5. The stub file is:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure6_1.png}
\caption{Benchmark empty}
\end{figure}

\textsuperscript{1}Department of Computing, Imperial College London
begin
n := 1;
while n > 0 do
  prev := n;
  n := 0;
  while prev > 0 do
    choose 2: n := n + 2 or 1: skip ro;
    prev := prev - 1
  od
od;
stop
end

Again, the declaration part varies to test performance as a function of number of states. The range of both variables starts at \{0..10\}, to finally reach \{0..450\}. The full population.ds file may again be found in the Appendix. This benchmark is supposed to represent a real program which one might write to investigate a phenomenon of interest. Since it uses only two variables, we did not expect the optimised implementation to outperform the regular one. We were curious however as to how would their performance differ, and most importantly, how would Julia compare to Octave on this practical benchmark.

![Figure 6.2: Benchmark population](image)

The results are shown in Figure 6.2. To improve readability, we present two plots: one of them compares Octave to averaged Julia implementations, and the other contrasts both Julia implementations. First of all, note that the plot in Figure 6.2a has some points missing - the lack of a mark indicates that the execution took longer than 1800 seconds, which is the timeout we set.

The results indicate that the Julia implementation outperforms Octave by orders of magnitude. Moreover, along with our expectations, the optimisations do not improve the performance on this benchmark; on the contrary, their overhead impacts it negatively.
Our next benchmark was created with a purpose of showing that the optimisations we implemented can improve the performance of our tool. Therefore, the stub (called many\_vars.pws) contains many variables, plenty of assignments and some conditionals:

```plaintext
begin
    x := 2 * a;
    if z > 0
        then b := z / 2
        else b := (-z) / 2
    fi;
    if x > 0
        then y := b + c - x
        else y := b + c + x
    fi;
    z := c - a;
end
```

As usual, variable ranges differ, starting from \(\{0,1\}\) for \(a, b\) and \(c\) and \(-2..2\) for \(x, y, z\), up to \(0..8\) and \(-16..16\) respectively. The results are shown in Figure 6.3.

Again, missing points denote that the timeout was reached. As expected, optimised Julia implementation outperforms the regular one on this benchmark. We also note that it computes the operators in a reasonable amount of time for number of states close to \(10^8\). This is impressive, since we saw from the first benchmark that roughly \(10^9\) states is a limitation imposed by the amount of memory in the machine on which our tests were run. So even for a relatively big program, the optimised Julia implementation comes close to that restriction.

Yet another benchmark (called multiplication) we devised is a program calculating the product of two integers using only addition and subtraction. In the below, \(a\) and \(b\) differ in order to test the performance as a function of state space dimension. Initially, they are both equal to 5, to finally reach 30. The code is as follows:

```
68
```
var
  r : {0..ab};
  a : {0..a};
  b : {0..b};
begin
  a ?= {0..a};
  b ?= {0..b};
  r := 0
  while a > 0 do
    r := r + b;
    a := a - 1;
  od;
  stop
end

We were curious whether the optimised Julia implementation would perform better than the regular one on this benchmark. The results are depicted in Figure 6.4.

We were glad to see that optimisations improve the performance of our tool on multiplication benchmark. Moreover, the difference in compilation times is significant. We therefore see that one does not have to invent artificially complex, meaningless programs such as many vars.pws to profit from our extra implementation effort (it could be argued that multiplication is artificially complex, but at least it computes something).

The general conclusion drawn from the presented results is that our implementation of pwc can successfully handle programs whose state space dimension does not exceed $10^7$. In some cases, even programs with close to $10^8$ states can be compiled efficiently. Moreover, we conclude that CPU performance is the main factor limiting the efficiency of our compiler, rather than the amount of RAM. Furthermore, optimisations should be used carefully – for some programs they might improve the performance significantly, while for the others they may aggravate it. The rule of thumb could be to enable them when the program has more than two variables and a few assignments or conditionals.

The plots suggest that the compilation time is increasing exponentially with the number of states. This could be surprising since the number of matrix operators is constant for a given program and the algorithm which computes each operator iterates over states and executes an approximately constant time operation.
in each. Some manual tests which we run have shown that the computation of update operators takes the longest, however we did not have enough time to investigate exactly which part of the algorithm is the bottleneck. Such an analysis could constitute a part of continuation of this project. Finally, our implementation outperforms the reference tool by orders of magnitude.
Chapter 7

Conclusions

During this project, we have investigated Linear Operator Semantics and produced a tool which puts its ideas into practice. We have presented a beautiful connection between discrete-time Markov chains and semantics of probabilistic programs (full credit for which goes to H. Wiklicky, A. Di Pierro and C. Hankin [4]). We have seen how abstract properties of Kronecker product can lead to drastic improvements in the performance of our compiler. Exhibiting this link between mathematics and computer science provides an excellent example of how the two areas are interconnected.

We have dealt with a variety of programming languages and paradigms. First of all, there is imperative \texttt{pWhile}, which we extended with new features. Majority of the code we have produced was written in a functional style in OCaml. We have used scripting features of Python to develop frameworks that enabled us to evaluate \texttt{pwc}. Finally, through Julia and Octave, we came into contact with numerical computing. We have shown the superiority of the former over the latter, and also its great potential for the future.

Using the developed tool, we could provide yet another proof of correctness of the security protocol presented as part of Dining Cryptographers problem and solve several probabilistic riddles.

7.1 Future work

Even though we completed the basic requirements of the project, we could not be fully satisfied with what we achieved. In particular, we believe a lot more can be done to improve performance of \texttt{pwc}. First of all, we note the parallelisation inherent to Linear Operator Semantics – the local operators corresponding to each block are independent of each other and could therefore be computed concurrently. Especially for programs with many blocks, this could result in a significant speed up. Moreover, Julia has built-in support for parallel computing, which could save one a lot of effort. Another way to increase the efficiency of our compiler is to run in on a better hardware. We have seen how the amount of RAM limits the size of programs \texttt{pwc} can handle. With more cores, one could
exploit parallel computation in a better way. Finally, specialised hardware, such as a GPU, could be used – that would require additional code to be written, but may well be worth the effort.

We have mentioned in Section 3.2.6 how pWhile could be extended further. This provides an almost endless list of improvements, which include not only adding new syntax, but also language features such as subroutines, modules, command line parameters, or even types and objects.

An interesting feature which could be added to \texttt{pwc} is detecting invalid states. In particular, when one of the variables is assigned a value out of its range as a result of executing a certain statement, the resulting state is not an element of \texttt{States} as defined before. Currently, this leads to the local operator having a row of zeros and hence losing its stochastic property. An alternative solution involves extending each local operator with an additional column. When the resulting state is invalid, the matrix entry in that column could be set to one. As a result, every local operator would be stochastic and therefore the state vector would also be stochastic at all times, with extra entries corresponding to invalid states.

Our tool could also be extended to support other semantics of probabilistic programs. Two examples of such can be found in a paper by Kozen [6], it would also be interesting to investigate whether LOS can be adapted to handle continuous-time Markov chains.

We have seen how replacing Octave with Julia brought along a huge performance gain. It could still be improved by optimising the code and introducing concurrent execution. However, it cannot be excluded that in five years time another high-performance numerical computing environment will emerge and outrun Julia’s performance. One might then want to implement \texttt{pwc}’s back end in this new language. In ten years time, there might appear yet another environment, better than the previous one. It would not be practical to reimplement the tool on every such occasion. To avoid having to do that, one could create a scripting language specifically targeted at requirements of Linear Operator Semantics. The compiler could then produce programs implemented in that new language, and they could be translated into any high-level programming language by supplying a suitable module with the syntax rules specific to that environment. This “translator” module would have to be implemented though and this task would be far from straightforward. Moreover, each language has its own characteristics and executes certain code patterns more efficiently than others. The automatically generated code would be generic however and manual effort would be required to optimise it. We therefore approach the described idea with some skepticism, however we believe it is worth investigating it further.
Bibliography


Appendix A

Code examples

A.1 Significant types

Below are some of the types referred to in Section 3.1.

• Types used to represent the declarations of a program

```ocaml
type range =
    int list
  ;;
type id =
    string
  ;;
type meta = (** metadata of a variable *)
  | Constant of int
  | Primitive of range
  | Array of int * range
  ;;
type decl =
    id * meta
  ;;
type decls =
    decl list
  ;;
```

• Different types of statements

```ocaml
type stmt =
  | Stop
```
A.2 Semantic analysis

The main function walks the syntax tree and verifies each statement on the way

```
let rec check_stmt stmt =
    match stmt with
    | Stop | Skip | Goto(_) ->
        ()
    | Tagged(t,s) ->
        check_stmt s
    | Assign(v,e) ->
        Expression.check_assigned_varref v;
        Expression.check_expr e
    | Random(v,r) ->
        Expression.check_assigned_varref v
    | Sequence(s1,s2) ->
        check_stmt s1;
        check_stmt s2
    | If(b,s1,s2) ->
        Expression.check_bexpr b;
        check_stmt s1;
```
check_stmt s2
| While(b,s) ->
  Expression.check_bexpr b;
  check_stmt s
| For(i,b,u,s) ->
  check_stmt i;
  Expression.check_bexpr b;
  check_stmt u;
  check_stmt s
| Case(a,csl,s) ->
  Expression.check_aexpr a;
  List.iter check_stmt (List.map snd csl);
  check_stmt s
| Repeat(s,b) ->
  check_stmt s;
  Expression.check_bexpr b
| Choose(wsl) ->
  List.iter check_stmt (List.map snd wsl)

The following code verifies that the LHS of an assignment is valid

let assign_primitive id meta =
  match meta with
  | Constant(_) ->
    failwith ("Can't assign to a constant " ^ id)
  | Primitive(_) -> ()
  | Array(_,_) ->
    failwith ("Index required when assigning to array " ^ id)

let assign_array id meta =
  match meta with
  | Constant(_) ->
    failwith ("Can't assign to a constant " ^ id)
  | Array(_,_) -> ()
  | Primitive(_) ->
    failwith ("Index not allowed when assigning to primitive " ^ id)

(** 'meta' returns the meta data associated to variable called 'id'
  let check_assigned_varref varref =
  let id = id varref in
  try
    let m = meta id in

    76
begin match varref with
  | Var(id) ->
    assign_primitive id m
  | ArrElem(id,e) ->
    assign_array id m
end
with Not_found ->
  failwith ("Variable " ^ id ^ " assigned to, but not declared")
;;

A.3 Refactor

We present a few code examples to support claims made in Section 4.2.

A.3.1 Commented code

- Julia function commented according to the official guidelines, which makes the comments accessible in the Julia REPL.

```julia
"""
swap!(arr, i, j)

Swaps elements at positions ‘i’ and ‘j’ in an array ‘arr’, in place. Assumes ‘i’ and ‘j’ are valid indexes into the array.
# Example
julia> arr = [1,2,3,4,5]
julia> swap!(arr,2,4)
julia> arr
5-element Array{Int64,1}:
  1
  4
  3
  2
  5
"""
```

```julia
function swap!(arr, i, j)
    @assert valid_index(length(arr), i) && valid_index(length(arr), j)
    if i != j
        arr[i], arr[j] = arr[j], arr[i]
    end
end
```

77
OCaml function commented according to the official guidelines, \texttt{ocamldoc}
can be used to generate documentation from these comments.

\begin{verbatim}
(**
  julia_blocks lblocks

  @param 'lblocks' list of labelled blocks
  (ie pairs (label, block))

  Main julia output function in this file, writes the number of
  blocks and update + filter operators along with corresponding
  test and assignment functions to the julia file.
*)
let julia_blocks lblocks =
  julia_separator ();
  julia_string "# Translation of blocks\n"
  julia_separator ();
  julia_blocks_number lblocks;
  julia_separator ();
  julia_helpers lblocks;
  julia_separator ();
  julia_string ("println("Compute state updates ")");
  julia_string ("and filter operators...")");
  julia_operators lblocks;

;
\end{verbatim}

\textbf{A.3.2 Extracting common functionality}

- Auxiliary functions for defining a function, assigning a value, applying a
  function and declaring dictionary in the generated Julia file.

\begin{verbatim}
let julia_function name args lines =
  julia_string "function ";
  julia_string name;
  julia_string "\n"
  julia_string (String.concat ", " args);
  julia_string "\n"
  julia_string (String.concat "\n" (List.map ((^) " ") lines));
\end{verbatim}
let julia_dict name entries =
    julia_string "const ";
    julia_string name;
    julia_string " = Dict(
	";
    julia_string (String.concat "
	" (List.map dict_entry entries));
    julia_string "\n)\n"
;;

• Some examples of functions above used in the code

let julia_ids2rng decls =
    julia_dict "id2rng" (List.map id2rng_entry decls);
;;

let random_assign x r l =
    let fl = "const F" ^ l in
    match x with
    | Var(id) -> julia_assignment fl (ur (id2ord id) r)
    | ArrElem(id,e) ->
        let size = string_of_int (size (meta id))
        and varrefs = aexpr_vars e in
        let ords = ordinals_julia_list varrefs in
        if !flagOpt
            then julia_assignment fl (ura_opt (id2ord id) ords size r l)
        else julia_assignment fl (ura (id2ord id) size r l)
    ;;

let julia_test_function l bexpr =
    let name = "test" ^ l
    and ret_val = apply_julia_func "convert"
        ["Int"; bexpr_to_julia_string bexpr]
A.3.3 Rewriting suboptimal functions

Consider the function \texttt{interval} which, given two integer arguments \(a\) and \(b\), returns a list \([a, a+1, \ldots, b]\). It was implemented in the reference tool in the following way:

\[
\begin{align*}
\text{let } \text{interval } a \ b &= \\
\text{let rec } \text{adduntil } lb \ il &= \\
&\quad \text{let } i = \text{List.hd } il \text{ in} \\
&\quad \text{if } lb \geq i \text{ (* lower bound } lb \text{ above head of } il \text{ *)} \\
&\quad \quad \text{then } il \text{ (* done, return integer list } il \text{ *)} \\
&\quad \quad \text{else adduntil } lb \ (i-1::il) \text{ (* put } i-1 \text{ in } il \text{ and continue *)} \\
\text{in} \\
&\quad \text{if } a>b \quad \text{(* empty interval *)} \\
&\quad \quad \text{else adduntil } a \ [b] \quad \text{(* add to interval starting with } [b] \text{ *)} \\
\end{align*}
\]


;;

We rewrote it as follows:

\[
\begin{align*}
\text{let rec } \text{interval } a \ b &= \\
&\quad \text{if } a>b \text{ then } [] \text{ else } a::(\text{interval } (a+1) \ b) \\
\end{align*}
\]


A.3.4 Handling output

We now present the new way of handling output, which uses the concept of \textit{default} and \textit{special} printing.

- Suppose binary expressions are represented by a following (simplified) variant type

\[
\begin{align*}
\text{type } \text{bexpr} &= \\
&\quad | \text{True} \\
&\quad | \text{False} \\
&\quad | \text{Not of bexpr} \\
&\quad | \text{And of bexpr } * \text{ bexpr} \\
&\quad | \text{Or of bexpr } * \text{ bexpr} \\
\end{align*}
\]

- We now want to specify different ways of printing binary expressions. To do that we must define an instance of \texttt{bexpr.printer}, which is a following record\(^1\)

\(^1\)C users might think of \texttt{record} as an ocaml’s equivalent of \texttt{struct}
type bexpr_printer =
  { print_true : unit -> string;
    print_false : unit -> string;
    print_not : string -> string;
    print_and : string -> string -> string;
    print_or : string -> string -> string;
  };

Note that the printing functions take strings rather than bexprs as arguments. Those arguments are assumed to be string representations of corresponding binary expressions.

• First, we must define a default way of printing binary expressions. We might declare a default_printer as follows (note that ^ is an infix string concatenation operator in ocaml)

```ocaml
let default_printer =
  let print_true () = "true"
  and print_false () = "false"
  and print_not e = "~" ^ e
  and print_and e1 e2 = e1 ^ "&&" ^ e2
  and print_or e1 e2 = e1 ^ "||" ^ e2
  in
  { print_true = print_true;
  print_false = print_false;
  print_not = print_not;
  print_and = print_and;
  print_or = print_or;
  }

;;
```

• Now suppose there is another probabilistic programming language called qWhile, which does not have a boolean type and so true and false are represented by 1 and 0 respectively. Moreover, and is only available as prefix operator and. We may then define qwhile_printer by 'overriding' appropriate functions of default_printer.

```ocaml
let qwhile_printer =
  let print_true () = "1"
  and print_false () = "0"
  and print_and e1 e2 = "and(" ^ e1 ^ ", " ^ e2 ^ ")"
  in
  { default_printer with
  print_true = print_true;
  ```
let rec bexpr_to_string ?(bp=default_bp) bexpr = match bexpr with
| True ->
  bp.print_true ()
| False ->
  bp.print_false ()
| Not(e) ->
  let e = bexpr_to_string ~bp:bp e in bp.print_not e
| And(e1,e2) ->
  let e1 = bexpr_to_string ~bp:bp e1 and e2 = bexpr_to_string ~bp:bp e2 in bp.print_and e1 e2
| Or(e1,e2) ->
  let e1 = bexpr_to_string ~bp:bp e1 and e2 = bexpr_to_string ~bp:bp e2 in bp.print_or e1 e2

A.3.5 Representing weight

We now compare two different ways in which weights can be represented – left column shows code snippets from the reference tool, right column presents our implementation

- Type definition

```plaintext
| CWeight of int |
| PWeight of string |

| int * int |
```

- Printing weight

```plaintext

| CWeight of int |
| PWeight of string |

| int * int |
```
let print_weight p =  
  match p with  
  | Statement.CWeight(w) ->  
    print_int w;  
  | Statement.PWeight(w) ->  
    print_string w;  

let print_weight (p,q) =  
  if q == 1 then  
    print_string ps  
  else begin  
    print_int ps;  
    print_string "/";  
    print_int q  
  end  

• Using weight in the code. This example illustrates the main weakness of the old representation.

octave_string "abs(";  
octave_weight p;  
octave_string ")*kron(F";  
octave_int i;  
match p with  
  | Statement.CWeight(w) ->  
    if w < 0  
    then octave_string "tt,"  
    else octave_string ",";  
  | Statement.PWeight(i) ->  
    octave_int 0  
(* should not happen *)

julia_string "abs(";  
Statement.julia_weight w;  
julia_string ") * kron(F";  
julia_int i;  
if fst w < 0  
then julia_string "t";

A.3.6 Variable references

Next, we compare how variable references and assignments are handled (columns as before)

• Types (irrelevant code is represented by (...))

  type varref =  
    | Var of id  
    | ArrElem of id * aexpr
  type stmt =  
    | Assign of aexpr * aexpr
    | Random of aexpr * range
(...)

83
The following function is used to generate the Octave code defining the local operators in the reference tool

```ocaml
let rec octave_transfer (l, b) = begin
  octave_string "global F";
  octave_int l;
  begin match b with
    | BStop -> (...)
    | BSkip -> (...)
    | BChoose -> (...)
    | BTest(b) -> (...)
    | BAsn(x, a) ->
      begin match x with
        | Var(xid) ->
          octave_string " = ";
          octave_asn l xid
        | Cell(aid, idx) ->
          octave_string " = ";
          octave_aasn l aid idx
        | _ ->
          failwith "Internal error (lhs of :=)\n"
      end
    | BRnd(x, r) -> (...)
  end;
  octave_string ";\n";
end
```

A similar function is defined as follows in our implementation (it was modified slightly for better presentation)

```ocaml
let julia_operator (l, blk) = begin
  julia_string "const F";
  julia_int l;
  begin match blk with
    | BStop -> (...)
    | BSkip -> (...)
    | BChoose -> (...)
    | BTest(b) -> (...)
    | BAsn(x, a) ->
      julia_string "=";
      begin match x with
        | Var(xid) ->
          julia_string " = ";
          julia_asn l xid
        | Cell(aid, idx) ->
          julia_string " = ";
          julia_aasn l aid idx
        | _ ->
          failwith "Internal error (lhs of :=)\n"
      end
    | BRnd(x, r) -> (...)
  end;
  julia_string ";\n";
end
```
A.4 LOS

This is the full LOS.jl file:

```julia
# LOS.jl

# This file contains functions used by julia files generated by pWhile compiler. Most of the naming conventions used in this file are inspired by the paper 'Probabilistic Semantics and Program Analysis' by A.D.Pierro, C.Hankin and H.Wiklicky. Some references to this paper are also made throught the comments in this code.

#--------------------------------------------------------------
# Auxiliary functions
#--------------------------------------------------------------

I(dim)
Return sparse identity matrix of dimension 'dim'

function I(dim)
    return speye(Int,dim)
end

E(m, n, i, j)
Return sparse 'm' by 'n' matrix with 'ij'-entry 1, others 0

function E(m, n, i, j)
    @assert valid_index(m, i) && valid_index(n, j)
    return speye(Int, m, n, i, j)
end
```

85
R = spzeros(Int, m, n)
R[i, j] = 1
return R
end

""
E(dim, i, j)
Return sparse square matrix with ‘ij’-entry 1, others 0,
dimension ‘dim’
""
function E(dim, i, j)
    @assert valid_index(dim, i) && valid_index(dim, j)
    R = spzeros(Int, dim, dim)
    R[i, j] = 1
    return R
end

""
E(dim, i)
Return sparse matrix with ‘ith diagonal entry 1, others 0,
dimension ‘dim’
""
function E(dim, i)
    return E(dim, i, i)
end

""
U_c(dim, c)
Return sparse square matrix of dimension ‘dim’ with entries
in ‘c’th column 1, otherwise 0
""
function U_c(dim, c)
    @assert valid_index(dim, c)
    R = spzeros(Int, dim, dim)
    for i = 1:dim
        R[i, c] = 1
    end
    return R
end

""
function e_i(dim, i)
    @assert valid_index(dim, i)
    R = spzeros(Int, 1, dim)
    R[i] = 1
    return R
end

# State of variables in a program can be represented twofold. 
# Firstly, it could be an array specifying value of each 
# variable (i.e. value of ith variable is at position i in 
# the array) - this is the representation described in the paper. 
# Alternatively, it could be represented by a single integer value 
# between 1 and d (inclusive), where d is the number of different 
# states. Now clearly there’s an isomorphism between the two 
# representations.

# Example
Suppose there are two variables, x and y, both can take 2 
# different values (it doesn’t matter what those values are). 
# Then the isomorphism is:
    """julia
    [[1,1] <-> 1
    [1,2] <-> 2
    [2,1] <-> 3
    [2,2] <-> 4
    """

index and unindex provide the mapping between the two 
representations.
- index transforms state represented by array of values 
  of variables into state represented by an integer 
- unindex does the opposite

# unindex(dims, i)

Return array representation of the state of the variables 
corresponding to the integer representation ‘i’
```julia
julia> unindex([2,2], 3)
2-element Array{Int64,1}:
2
1
```

```julia
function unindex(dims::Array{Int,1}, i::Int)
    @assert i <= prod(dims)
    result::Array{Int,1} = []
    if length(dims) > 1
        result = unindex(dims[1:end-1], div(i - 1, dims[end]) + 1)
    end
    push!(result, rem(i - 1, dims[end]) + 1)
    return result
end

index(dims, values)
```

Return integer representation of the state of the variables
 corresponding to the array representation ‘values’

```julia
julia> index([2,2], [2,1])
3
```

```julia
function index(dims::Array{Int,1}, values::Array{Int,1})
    @assert length(dims) == length(values)
    for i=1:length(dims)
        @assert dims[i] >= values[i]
    end
    if length(dims) == 1
        return values[1]
    else
        return (values[1] - 1) * prod(dims[2:end]) +
        index(dims[2:end], values[2:end])
    end
end
```
valid_index(len, i)

Check whether ‘i’ is a valid index into a vector of length ‘len’

# Example
c"""julia
julia> valid_index(5,2)
true
c"""

julia> valid_index(5,0)
false
c"""

function valid_index(len, i)
    return i > 0 && i <= len
end

swap!(arr, i, j)

Swaps elements at positions ‘i’ and ‘j’ in an array ‘arr’, in place. Assumes ‘i’ and ‘j’ are valid indexes into the array.

# Example
c"""julia
julia> arr = [1,2,3,4,5]
julia> swap!(arr,2,4)
julia> arr
5-element Array{Int64,1}:
   1
   4
   3
   2
   5
c"""

function swap!(arr, i, j)
    @assert valid_index(length(arr), i) &&
        valid_index(length(arr), j)
    if i != j
        arr[i], arr[j] = arr[j], arr[i]
    end
end
Given ‘ordinals’ (of variables that appear in an arithmetic expression) compute how variables need to be swapped to minimise the number of swaps. Using the fact that variables will be swapped into positions 1,2,...,n, where n is length of ‘ordinals’, the swaps are represented as an array of length n.

```julia
julia> compute_swaps([7,1,6,4])
4-element Array{Int64,1}:
    1
    7
    6
    4
```

Here the resulting array represents the swaps:

1 and 4 are in brackets since they don’t represent actual swaps

```julia
function compute_swaps(ordinals::Array{Int,1})
    result = sort(union(ordinals))
    n = length(result)
    for i = n:-1:1
        if result[i] <= n && result[i] != i
            swap!(result, i, result[i])
        end
    end
    return result
end
```

Given variable dimensions in the initial order ‘dims’, compute the dimensions order after the swap represented by ‘swaps’ (so returned value is a permutation of ‘dims’)

```julia
compute_swapped_dims(dims, swaps)
```
function compute_swapped_dims(dims::Array{Int,1},
                          swaps::Array{Int,1})
    result = copy(dims)
    for i = 1:length(swaps)
        swap!(result, i, swaps[i])
    end
    return result
end

function extend(values_restricted::Array{Int,1},
                swaps::Array{Int,1}, n::Int)
    for i in swaps
        @assert i <= n
        values_restricted[i] = 2
    end
    return values_restricted
end

# Example
`julia
julia> compute_swapped_dims([3,3,2,2], [3,4])
4-element Array{Int64,1}:
  2
  2
  3
  3
`
function compute_swap_operator(dims::Array{Int,1}, p::Int, q::Int)
    @assert valid_index(length(dims), p) && valid_index(length(dims), q)
    d = prod(dims)
    if p == q
        return I(d)
    end
    R = spzeros(Int, d, d)
    for i = 1:dims[p]
        for j = 1:dims[q]
            A = I(1)
            for k = 1:p-1
                A = kron(A, I(dims[k]))
            end
            A = kron(A, E(dims[p], dims[q], i, j))
            for k = p+1:q-1
                A = kron(A, I(dims[k]))
            end
            A = kron(A, transpose(E(dims[p], dims[q], i, j)))
            for k = q+1:length(dims)
                A = kron(A, I(dims[k]))
            end
            R += A
        end
    end
    return R
end

Compute operator (i.e. matrix) that swaps ‘p’th and ‘q’th variables
It is based on an idea of a commutator matrix which makes Kronecker
product commute.
""" compute_swaps_operator(dims, swaps) Compute operator (i.e. matrix) that swaps variables as represented by ‘swaps’ vector. It is the product of swap operators for each individual swap.

"""

function compute_swaps_operator(dims::Array{Int,1},
                               swaps::Array{Int,1})
    R = I(prod(dims))
    for i = 1:length(swaps)
        if i != swaps[i]
            R = R * compute_swap_operator(dims, i, swaps[i])
        end
    end
    return R
end

#--------------------------------------------------------------
# State update operators and filter operators
#--------------------------------------------------------------
#
# In all functions below dims is the vector of variable dimensions
#
"""

U_xk_c(dims, k, c)

Constant update operator

Compute operator that assigns ‘k’th variable value ‘c’ (i.e. the ‘c’th value in its range - NOT RIGHT NOW), and leaves others as they are

"""

function U_xk_c(dims::Array{Int,1}, ordinal::Int, c::Int)
    R = I(1)
    for i = 1:(ordinal - 1)
        R = kron(R, I(dims[i]))
    end
    if c > 0
        R = kron(R, U_c(dims[ordinal], c))
    else
        ...
    end
end
R = kron(R, spzeros(Int, dims[ordinal], dims[ordinal]))
end
for i = (ordinal + 1):length(dims)
    R = kron(R, I(dims[i]))
end
return R
end

#--------------------------------------------------------------
# Update operators
#--------------------------------------------------------------

""
Ue(dims, ordinal, update)

Variable update operator

Compute the operator that assigns to variable with ordinal
'ordinal' the value of expression e, where the value, given
values of variables, is computed using the function 'update'.
This is the basic implementation which doesn’t take into
account the number of variables appearing on the RHS of the
assignment. It loops over all the possible combinations of
variable values (and uses 'unindex' to recover those) and
computes the value of the expression (using the 'update'
function) for those values of variables.

""
function Ue(dims::Array{Int,1}, ordinal::Int, update::Function)
    @assert ordinal <= length(dims)
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 1:d
        values = unindex(dims, i)
        values[ordinal] = update(values)
        if values[ordinal] > 0
            R[i,index(dims, values)] = 1
        end
    end
    return R
end

""
Ue(dims, ordinal, ordinals, update)
Optimised variable update operator

Compute the operator that assigns to variable with ordinal ‘ordinal’ the value of expression e, where the value, given values of variables, is computed using the function ‘update’. ‘ordinals’ is the vector of ordinals of variables that appear on the RHS of the assignment. With that, the operator can be computed more efficiently by swapping involved variables so they come first, compute the smaller update operator restricted to involved variables and compute Kronecker product of that operator with identity matrices representing lack of influence of other variables on the current assignment.

Special case is when ‘ordinals’ is empty - it means the expression on the RHS of the assignment does not contain any variables, i.e. has a numeric value. Hence constant update operator from above can be used.

```julia
function Ue(dims::Array{Int,1}, ordinal::Int,
    update::Function, ordinals::Array{Int,1})
    if length(ordinals) == 0
        values = ones(Int, 1, length(dims))
        return U_xk_c(dims, ordinal, update(values))
    end
    swaps = compute_swaps(push!(copy(ordinals), ordinal))
    n = length(swaps)
    dims_swapped = compute_swapped_dims(dims, swaps)
    dims_restricted = dims_swapped[1:n]
    newordinal = findfirst(swaps, ordinal)
    d = prod(dims_restricted)
    R = spzeros(Int,d,d)
    for i = 1:d
        values_restricted = unindex(dims_restricted, i)
        values = extend(values_restricted, swaps, length(dims))
        values_restricted[newordinal] = update(values)
        if values_restricted[newordinal] > 0
            R[i, index(dims_restricted, values_restricted)] = 1
        end
    end
    for dim in dims_swapped[n+1:end]
        R = kron(R,I(dim))
    end
end
```
K = compute_swaps_operator(dims, swaps)
return K * R * K
end

# Array update operators
#--------------------------------------------------------------

#--------------------------------------------------------------

Ua_c(dims, ordinal, compute_index, size, c)

Constant array update operator

Compute operator that assigns value ‘c’ to the kth element of the array (of size ‘size’) that starts at ordinal ‘ordinal’, where k is computed given values of all variables by function ‘compute_index’.

function Ua_c(dims::Array{Int,1}, ordinal::Int,
              compute_index::Function, size::Int, c::Int)
    @assert ordinal <= length(dims)
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 1:d
        values = unindex(dims, i)
        pos = compute_index(values)
        if valid_index(size, pos + 1)
            values[ordinal + pos] = c
            if values[ordinal + pos] > 0
                R[i,index(dims, values)] = 1
            end
        end
    end
    return R
end

Ua_c(dims, ordinal, compute_index, index_ordinals, size, c)

Optimised constant array update operator

As above, but uses extra information in ‘index_ordinals’ which tells which variables appear in the index expression.
function Ua_c(dims::Array{Int,1}, ordinal::Int,
    compute_index::Function, index_ordinals::Array{Int,1},
    size::Int, c::Int)
    @assert ordinal <= length(dims)
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 0:size-1
        R += P(dims, compute_index, index_ordinals, i) *
            U_xk_c(dims, ordinal + i, c)
    end
    return R
end

function Ua(dims::Array{Int,1}, ordinal::Int,
    compute_index::Function, index_ordinals::Array{Int,1},
    size::Int, update::Function,
    update_ordinals::Array{Int,1})
    @assert ordinal <= length(dims)
    if length(index_ordinals) == 0
        index = compute_index([])
        if valid_index(size, index + 1)
            return Ue(dims, ordinal + index, update, update_ordinals)
        end
    end
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 0:size-1
        R += P(dims, compute_index, index_ordinals, i) *
            update(dims, ordinal + i, update) *
            U_xk_c(dims, ordinal + i, c)
    end
    return R
end

Optimised array update operator

Compute the operator that assigns to nth element of an array of size ‘size’, whose first element has ordinal ‘ordinal’, value given by function ‘update’, where n is given by function ‘compute_index’ (which, the same as ‘update’, takes as argument values of all variables, so n differs depending on the values of variables). Uses the extra information in ‘index_ordinals’ - which variables appear in the index expression, and ‘update_ordinals’ - which variables appear in the RHS expression.

function Ua(dims::Array{Int,1}, ordinal::Int,
    compute_index::Function, index_ordinals::Array{Int,1},
    size::Int, update::Function,
    update_ordinals::Array{Int,1})
    @assert ordinal <= length(dims)
    if length(index_ordinals) == 0
        index = compute_index([])
        if valid_index(size, index + 1)
            return Ue(dims, ordinal + index, update, update_ordinals)
        end
    end
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 0:size-1
        R += P(dims, compute_index, index_ordinals, i) *
            U_xk_c(dims, ordinal + i, c)
    end
    return R
end

---
Ue(dims, ordinal + i, update, update_ordinals)
end
return R
end

"""
Ua(dims, ordinal, compute_index, size, update)

Array update operator

Compute the operator that assigns to nth element of an array of size 'size', whose first element has ordinal 'ordinal',
value given by function 'update', where n is given by function
'compute_index' (which, the same as 'update', takes as argument
values of all variables, so n differs depending on the values
of variables)
"""

function Ua(dims::Array{Int,1}, ordinal::Int,
    compute_index::Function, size::Int, update::Function)
    @assert ordinal <= length(dims)
    d = prod(dims)
    R = spzeros(Int,d,d)
    for i = 1:d
        values = unindex(dims, i)
        pos = compute_index(values)
        if valid_index(size, pos + 1)
            values[ordinal + pos] = update(values)
            if values[ordinal + pos] > 0
                R[i,index(dims, values)] = 1
            end
        end
    end
    return R
end

#--------------------------------------------------------------
# Filter operators
#--------------------------------------------------------------

"""
P(dims, ordinals, test, c)

Optimised filter operator
Compute the filter operator filtering states in which expression 'e' evaluates to value 'c', where 'c' is 'true' or 'false'. Function 'test' is used to evaluate the expression e given values of all variables. It follows similar pattern as the optimised update operator, i.e. it uses the vector of ordinals of variables that appear in the expression 'e' to restrict the computation of the filter operator to only those variables and then naturally extend it to other variables using Kronecker product. Special case here is when no variables appear in expression 'e' - then the operator is simply identity when 'e' is true and zero matrix when 'e' is false.

```plaintext
function P(dims::Array{Int,1}, test::Function, ordinals::Array{Int,1}, c::Int)
    if length(ordinals) == 0
        values = squeeze(ones(Int,1,length(dims)),1)
        d = prod(dims)
        if test(values) == c
            return I(d)
        else
            return spzeros(Int,d,d)
        end
    end
    swaps = compute_swaps(ordinals)
    n = length(swaps)
    dims_swapped = compute_swapped_dims(dims, swaps)
    dims_restricted = dims_swapped[1:n]
    d = prod(dims_restricted)
    R = spzeros(Int,d,d)
    for i = 1:d
        values_restricted = unindex(dims_restricted, i)
        values = extend(values_restricted, swaps, length(dims))
        if test(values) == c
            R[i,i] = 1
        end
    end
    for dim in dims_swapped[n+1:end]
        R = kron(R,I(dim))
    end
    K = compute_swaps_operator(dims, swaps)
    return K * R * K
end
```
Filter operator

Compute the filter operator filtering states in which expression e evaluates to value ‘c’, where ‘c’ is ‘true’ or ‘false’. Function ‘test’ is used to evaluate the expression e given values of all variables.

```
function P(dims::Array{Int,1}, test::Function, c::Int)
d = prod(dims)
R = spzeros(Int,d,d)
for i = 1:d
    values = unindex(dims, i)
    if test(values) == c
        R[i,i] = 1
    end
end
return R
end
```

#--------------------------------------------------------------
# Auxiliary functions used to analyse the state vector
#--------------------------------------------------------------

#= The state vector of a given pWhile program very often has large
dimensions and quite a few non zero entries. It is usually
hard to deduce anything form it by just looking at it.
However, one frequently needs to know the probabilities
of some variables having certain values. The 'find_probability'
function enables one to compute that easily. .
=#

function match(values::Array{Int,1}, expected_values::Array{Int,1})
    @assert length(values) == length(expected_values)
    for i = 1:length(values)
        if expected_values[i] != 0 && expected_values[i] != values[i]
            return false
        end
    end
    return true
end
return true
end

""

find_probability(dims, block, state_vector, expected_values)

Retrieve the probability of variables having values given
by ‘expected_values’ given ‘state_vector’

As usual, ‘dims’ is an array holding ordered dimensions of
variables. ‘state_vector’ is the distribution of the
probabilistic state of the program. Finally, ‘expected_values’
is an array specifying expected values of variables (where as
usual values are given as indices into variable’s range). The
length of this array should be the same as the length of ‘dims’
and each entry should be smaller or equal than the corresponding
entry in ‘dims’ (since a variable must take a value in its range). Moreover, expected value of any variable can be set to 0 to denote
that this variable may take any value in its range (this is useful
since in most cases we only care about some subset of variables)

# Example

```julia
julia> find_probability([2, 2], [1//2, 0, 0, 0, 1//2, 0, 0, 0], [1, 1])
1//2
julia> find_probability([2, 2], [1//5, 0, 1//5, 1//5, 2//5, 0, 0, 0], [1, 0])
3//5
```

""

def function find_probability(dims::Array{Int,1}, state_vector,
    expected_values::Array{Int,1})

d = prod(dims)
blocks = div(length(state_vector), d)
result = 0
for i = 1:d
    values = unindex(dims, i)
    if match(values, expected_values)
        for j = 1:blocks
            result += state_vector[(i-1) * blocks + j]
        end
    end
end
return result
Below are two example Julia files which define the matrix operators for some pWhile programs and use functions defined in LOS.jl

- This one does not use optimisations

```julia
include("../LOS.jl") # Linear operator semantics
#------------------------------------------------------------
# Generated from population.pw by pWhile compiler -
(c) 2016 H.Wiklicky, M.Olejnik
#------------------------------------------------------------

const id2rng = Dict(
    "n" => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    "prev" => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
)

#------------------------------------------------------------
const id2ord = Dict(
    "n" => 1,
    "prev" => 2
)

#------------------------------------------------------------
const ord2rng = Dict(
    1 => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    2 => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
)

#------------------------------------------------------------
const v = length(ord2rng) # number of variables

dims = Array{Int}(v)
for i=1:v dims[i] = length(ord2rng[i]) end

const d = prod(dims)
```

102
const b = 10 # number of blocks

function assign1(values)
    return findfirst(ord2rng[id2ord["n"]], 1)
end

function test2(values)
    return convert(Int, (id2rng["n"][values[id2ord["n"]]>0])
end

function assign3(values)
    return findfirst(ord2rng[id2ord["prev"],
    id2rng["n"]][values[id2ord["n"]]])
end

function assign4(values)
    return findfirst(ord2rng[id2ord["n"]], 0)
end

function test5(values)
    return convert(Int,
    (id2rng["prev"])[values[id2ord["prev"]]>0])
end

function assign7(values)
    return findfirst(ord2rng[id2ord["n"]],
    (id2rng["n"])[values[id2ord["n"]]+2])
end
function assign9(values)
    return findfirst(ord2rng[id2ord["prev"]],
                     (id2rng["prev"][values[id2ord["prev"]]]-1))
end

println("Compute state updates and filter operators...")

const F1 = Ue(dims, id2ord["n"], assign1)
const F2 = F2f
const F3 = Ue(dims, id2ord["prev"], assign3)
const F4 = Ue(dims, id2ord["n"], assign4)
const F5t = P(dims, test5, 1)
const F5f = P(dims, test5, 0)
const F5 = F5f
const F6 = I(d)
const F7 = Ue(dims, id2ord["n"], assign7)
const F8 = I(d)
const F9 = Ue(dims, id2ord["prev"], assign9)
const F10 = I(d)

#------------------------------------------------------------

# Translation of flow

println("Compute transfer operators and Markov operator...")

const T12 = abs(1) * kron(F1, E(b, 1, 2))
const T23 = abs(-1) * kron(F2t, E(b, 2, 3))
const T34 = abs(1) * kron(F3, E(b, 3, 4))
const T45 = abs(1) * kron(F4, E(b, 4, 5))
const T56 = abs(-1) * kron(F5t, E(b, 5, 6))
const T67 = abs(2//3) * kron(F6, E(b, 6, 7))
const T68 = abs(1//3) * kron(F6, E(b, 6, 8))
const T79 = abs(1) * kron(F7, E(b, 7, 9))
const T89 = abs(1) * kron(F8, E(b, 8, 9))
const T95 = abs(1) * kron(F9, E(b, 9, 5))
const T52 = abs(1) * kron(F5, E(b, 5, 2))
const T210 = abs(1) * kron(F2, E(b, 2, 10))
const T1010 = abs(1) * kron(F10, E(b, 10, 10))

#------------------------------------------------------------
const T = T12 + T23 + T34 + T45 + T56 + T67 + T68 +
         T79 + T89 + T95 + T52 + T210 + T1010

# Translation of flow finished
println("Done")

• This one uses optimisations

include("../LOS.jl") # Linear operator semantics
#------------------------------------------------------------

# Generated from monty.pw by pWhile compiler -
(c) 2016 H.Wiklicky, M.Olejnik

#------------------------------------------------------------

const id2rng = Dict(
    "d" => [0, 1, 2],
    "g" => [0, 1, 2],
    "o" => [0, 1, 2]
)

#------------------------------------------------------------

const id2ord = Dict(
    "d" => 1,
    "g" => 2,
    "o" => 3
)

#------------------------------------------------------------

const ord2rng = Dict(
    1 => [0, 1, 2],
    2 => [0, 1, 2],
    3 => [0, 1, 2]
)
const v = length(ord2rng) # number of variables
dims = Array{Int}(v)
for i=1:v dims[i] = length(ord2rng[i]) end
const d = prod(dims)

# Translation of declarations finished

# Translation of blocks

const b = 9 # number of blocks

function test4(values)
    return convert(Int,
        (id2rng["o"][values[id2ord["o"]]]==
        id2rng["g"] [values[id2ord["g"]]]) ||
        (id2rng["o"] [values[id2ord["o"]]]==
        id2rng["d"] [values[id2ord["d"]]]))
end

function assign5(values)
    return findfirst(ord2rng[id2ord["o"]],
        ((id2rng["o"] [values[id2ord["o"]]] + 1) % 3))
end

function assign6(values)
    return findfirst(ord2rng[id2ord["g"]],
        ((id2rng["g"] [values[id2ord["g"]]] + 1) % 3))
end
function test7(values)
    return convert(Int,
           (id2rng["g"] [values[id2ord["g"]]]==
            id2rng["o"] [values[id2ord["o"]]]))
end

function assign8(values)
    return findfirst(ord2rng[id2ord["g"]],
                      ((id2rng["g"] [values[id2ord["g"]]]+1)%3))
end

#-----------------------------------------------------------
println("Compute state updates and filter operators...")

const F1 = 1//3*
    (U_xk_c(dims, id2ord["d"],
       findfirst(ord2rng[id2ord["d"]], 0)) +
     U_xk_c(dims, id2ord["d"],
       findfirst(ord2rng[id2ord["d"]], 1)) +
     U_xk_c(dims, id2ord["d"],
       findfirst(ord2rng[id2ord["d"]], 2)))

const F2 = 1//3*
    (U_xk_c(dims, id2ord["g"],
       findfirst(ord2rng[id2ord["g"]], 0)) +
     U_xk_c(dims, id2ord["g"],
       findfirst(ord2rng[id2ord["g"]], 1)) +
     U_xk_c(dims, id2ord["g"],
       findfirst(ord2rng[id2ord["g"]], 2)))

const F3 = 1//3*
    (U_xk_c(dims, id2ord["o"],
       findfirst(ord2rng[id2ord["o"]], 0)) +
     U_xk_c(dims, id2ord["o"],
       findfirst(ord2rng[id2ord["o"]], 1)) +
     U_xk_c(dims, id2ord["o"],
       findfirst(ord2rng[id2ord["o"]], 2)))

const F4t = P(dims, test4, round(Int, [id2ord["o"],
                                         id2ord["g"],
                                         id2ord["o"],
                                         id2ord["d"]]), 1)
const F4f = P(dims, test4, round(Int, [id2ord["o"],
                                         id2ord["g"],
                                         id2ord["o"],
                                         id2ord["d"]]), 1)
id2ord["g"], id2ord["o"], id2ord["d")), 0)
const F4 = F4f
const F5 = Ue(dims, id2ord["o"], assign5,
    round(Int, [id2ord["o"]]))
const F6 = Ue(dims, id2ord["g"], assign6,
    round(Int, [id2ord["g"]]))
const F7t = P(dims, test7,
    round(Int, [id2ord["g"], id2ord["o"]]), 1)
const F7f = P(dims, test7,
    round(Int, [id2ord["g"], id2ord["o"]]), 0)
const F7 = F7f
const F8 = Ue(dims, id2ord["g"], assign8,
    round(Int, [id2ord["g"]]))
const F9 = I(d)

#-----------------------------------------------------------
# Translation of flow
#-----------------------------------------------------------
println("Compute transfer operators and Markov operator...")
const T12 = abs(1) * kron(F1, E(b, 1, 2))
const T23 = abs(1) * kron(F2, E(b, 2, 3))
const T34 = abs(1) * kron(F3, E(b, 3, 4))
const T45 = abs(-1) * kron(F4t, E(b, 4, 5))
const T54 = abs(1) * kron(F5, E(b, 5, 4))
const T46 = abs(1) * kron(F4, E(b, 4, 6))
const T67 = abs(1) * kron(F6, E(b, 6, 7))
const T78 = abs(-1) * kron(F7t, E(b, 7, 8))
const T87 = abs(1) * kron(F8, E(b, 8, 7))
const T79 = abs(1) * kron(F7, E(b, 7, 9))
const T99 = abs(1) * kron(F9, E(b, 9, 9))

#-----------------------------------------------------------
const T = T12 + T23 + T34 + T45 + T54 + T46 +
            T67 + T78 + T87 + T79 + T99
println("Done")
A.5 Performance testing framework

A.5.1 TestFiles

This section presents all the programs (stub + declarations) used to evaluate the performance of our compiler.

- The first program is essentially empty. The stub is

```plaintext
# file empty.pws
begin
stop
end
```

and the declarations are

```plaintext
#file empty.ds
var
  x : {0..10};
y : {0..10};
z : {0..10};
or
var
  x : {0..100};
y : {0..100};
z : {0..10};
or
var
  x : {0..100};
y : {0..100};
z : {0..100};
or
var
  x : {0..1000};
y : {0..100};
z : {0..100};
or
```

109
var
  x : {0..1000};
y : {0..1000};
z : {0..100};
or
var
  x : {0..1000};
y : {0..1000};
z : {0..1000};

- The next program is more interesting:

```pws
# file populations.pws
begin
  n := 1;
  while n > 0 do
    prev := n;
    n := 0;
    while prev > 0 do
      choose 2/3: n := n + 2 or 1/3: skip
    od
  od;
stop
end
```

and the declarations

```ds
# file populations.ds
var
  n : {0..100};
  prev : {0..10};
or
var
  n : {0..100};
  prev : {0..100};
or
var
  n : {0..1000};
  prev : {0..1000};
```
or

```plaintext
var
  n : {0..10000};
  prev : {0..1000};
or
var
  n : {0..10000};
  prev : {0..10000};
```

- Benchmark many_vars:

```plaintext
# file many_vars.pws
begin
  x := 2 * a;
  if z > 0 then b := z / 2 else b := (-z) / 2 fi;
  if x > 0 then y := b + c - x else y := b + c + x fi;
  z := c - a;
stop
end
```

and the declarations

```plaintext
var
  a : {0..1};
  b : {0..1};
  c : {0..1};
  x : {-2..2};
  y : {-2..2};
  z : {-2..2};
or
var
  a : {0..2};
  b : {0..2};
  c : {0..2};
  x : {-4..4};
  y : {-4..4};
  z : {-4..4};
or
var
  a : {0..3};
  b : {0..3};
  c : {0..3};
  x : {-6..6};
  y : {-6..6};
```
\begin{verbatim}
z : {-6..6};
\end{verbatim}
or
\begin{verbatim}
var
  a : {0..4};
  b : {0..4};
  c : {0..4};
  x : {-8..8};
  y : {-8..8};
  z : {-8..8};
\end{verbatim}
or
\begin{verbatim}
var
  a : {0..6};
  b : {0..6};
  c : {0..6};
  x : {-12..12};
  y : {-12..12};
  z : {-12..12};
\end{verbatim}
or
\begin{verbatim}
var
  a : {0..8};
  b : {0..8};
  c : {0..8};
  x : {-16..16};
  y : {-16..16};
  z : {-16..16};
\end{verbatim}

\section{A.5.2 Report}

Test performance on stub file empty

Stub is:
begin
stop
end

Configuration 1:

The declarations are as follows:
\begin{verbatim}
var
  x : {0..10};
  y : {0..10};
  z : {0..10};
\end{verbatim} Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Average time: 0.0756121
Execute julia performance script
Average time: 0.432773
Execute julia performance script optimised
Average time: 0.432643

Configuration 2:

The declarations are as follows:

```
var
  x : \{0..100\};
  y : \{0..10\};
  z : \{0..10\};
```

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Average time: 0.0495019
Execute julia performance script
Average time: 0.463004
Execute julia performance script optimised
Average time: 0.461709

(...)

Configuration 6:

The declarations are as follows:

```
var
  x : \{0..1000\};
  y : \{0..1000\};
  z : \{0..100\};
```

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Average time: 5.25507
Execute julia performance script
Average time: 3.981792
Execute julia performance script optimised
Average time: 3.955252

Configuration 7:

The declarations are as follows:

```plaintext
var
  x : {0..1000};
  y : {0..1000};
  z : {0..1000};
```

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Script execution failed: error: out of memory or dimension too large for Octave’s index type
error: called from:
error:  /usr/share/octave/3.8.1/m/sparse/speye.m at line 50, column 5
error:  /homes/mo1712/year4/pwc/performance/generated/empty7.m at line 119, column 1
Execute julia performance script
Script execution failed:ERROR: LoadError: OutOfMemoryError()
  in call at ./essentials.jl:200
  in speye at sparse/sparsematrix.jl:597
  in I at /homes/mo1712/year4/pwc/performance/generated/../LOS.jl:22
  in include at ./boot.jl:261
  in include_from_node1 at ./loading.jl:320
  in process_options at ./client.jl:257
  in _start at ./client.jl:378
while loading /homes/mo1712/year4/pwc/performance/generated/empty7.jl
  , in expression starting on line 62
Execute julia performance script optimised
Script execution failed:ERROR: LoadError: OutOfMemoryError()
  in call at ./essentials.jl:200
  in speye at sparse/sparsematrix.jl:597
  in I at /homes/mo1712/year4/pwc/performance/generated/../LOS.jl:22
  in include at ./boot.jl:261
  in include_from_node1 at ./loading.jl:320
Test performance on stub file population

Stub is:
begin
n := 1;
while n > 0 do
    prev := n;
    n := 0;
    while prev > 0 do
        choose 2: n := n + 2 or 1: skip ro;
        prev := prev - 1
    od
od;
stop
end

Configuration 1:
The declarations are as follows:
var
    n : {0..10};
    prev : {0..10};
Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Average time: 1.14195
Execute julia performance script
Average time: 1.077369
Execute julia performance script optimised
Average time: 1.334574

Configuration 2:
The declarations are as follows:
var n : {0..30};
prev : {0..30};

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS

Execute octave performance script
Average time: 20.607

Execute julia performance script
Average time: 1.173089

Execute julia performance script optimised
Average time: 1.429143

Configuration 3:
The declarations are as follows:
var n : {0..50};
prev : {0..50};

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS

Execute octave performance script
Average time: 89.0059

Execute julia performance script
Average time: 1.350591

Execute julia performance script optimised
Average time: 1.712779

Configuration 9:
The declarations are as follows:
var n : {0..190};
prev : {0..190};

Compile the file using octave pwc...SUCCESS
Compile the file using julia pwc...SUCCESS
Compile the file using julia pwc with optimisations...SUCCESS

Execute octave performance script
Average time: 20.607

Execute julia performance script
Average time: 1.173089

Execute julia performance script optimised
Average time: 1.429143
Compile the file using julia pwc with optimisations...SUCCESS
Execute octave performance script
Timeout, aborting.

Execute julia performance script
Average time: 40.416081
Execute julia performance script optimised
Average time: 54.577166