Universiteit
Leiden

Master Computer Science

Formal Semantics of ALCH

Name: Loes Dekker
Student ID: s1850024

Date: 07/07/2021

Specialisation: Foundations of Computing

1st supervisor: Dr. Henning Basold
2nd supervisor: Prof.dr. Jetty Kleijn

Master's Thesis in Computer Science

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

Abstract

We present formal semantics for the Algorithmic Language for Chemistry (ALCH) [1], an imperative language with classical programming constructs that implements Chemical Reaction Network-Controlled Tile Assembly Models (CRN-TAM) [2]. We define the operational and denotational semantics. These are based on the sub-distribution monad, which models partial probability distributions. This monad provides a convenient setting for compositional semantics, as its associated Kleisli category allows to give compositional semantics to non-terminating probabilistic programs. The biggest challenge in defining the semantics is a probabilistic branching construct of the language, which may induce non-terminating backtracking of reversible code.

Contents

1 Introduction **4**

2 Background **5**
2.1 CRN-TAM . . . 5
2.2 ALCH . . . 7
2.3 Some Category Theory . . . 9

3 Related Work **12**

4 Operational Semantics **13**
4.1 Classical Constructs . . . 13
4.2 CRN-TAM Constructs . . . 14
4.3 The Branch Construct . . . 19
4.3.1 Branch Paths and the Backtracking Relation . . . 19
4.3.2 Semantics of the Branch Expression . . . 21
4.4 Overview of the Operational Semantics . . . 23

5 Denotational Semantics **25**
5.1 Classical Constructs . . . 25
5.2 CRN-TAM Constructs . . . 25
5.3 Some More Monads . . . 26
5.4 The Branch Construct . . . 28
5.5 Overview of the Denotational Semantics . . . 30
5.6 Examples . . . 31

6 Conclusion **35**

7 Acknowledgements **35**

1 Introduction

The Algorithmic Language for Chemistry (ALCH) is a high-level imperative language that implements a model for self-assembly [1]. Winfree introduced the first model for DNA tile based self-assembly in 1998 and proved that, theoretically, self-assembly is capable of Turing-universal computing [3]. In the following years Winfree's abstract Tile Assembly Model (aTAM) has been expanded on in numerous ways, for example to make it more realistic with regards to actual experiments [4]. ALCH implements a model which combines elements of aTAM and Chemical Reaction Networks (CRN), called the Chemical Reaction Network-Controlled Tile Assembly Model (CRN-TAM). CRN-TAM is a probabilistic self-assembly model [2], whereas aTAM is deterministic.
The outcome of an ALCH program can be described as sub-probability distribution over program states, which can incidentally be embedded in a categorical construct. This thesis presents a formal semantics which uses categorical constructs to define the meaning of ALCH programs. Formal semantics can be used to show program equivalence for example. The meaning of ALCH programs will be defined both using operational semantics and denotational semantics. Operational semantics gives programs meaning by defining an abstract machine, where the program state is represented by a simple memory, and specifying rules which determine what actions this abstract machine can execute [5]. Operational semantics allow the user to grasp how a program progresses. Denotational semantics give the program meaning as a mathematical function by assigning a meaning to every construct in the language and composing these [5]. It is mainly in the denotational semantics that category theory are used.

ALCH consists of both classical programming constructs, which are deterministic, and CRN-TAM constructs, which are probabilistic. In this thesis, the classical constructs will be awarded a formal semantics first, before extending these to the probabilistic constructs. This approach was described in Churchill's dissertation [6], in which formal semantics are defined for a probabilistic quantum language.
In Section 2 a surface level explanation of self-assembly is given, so that CRN-TAM and its predecessors can be properly introduced. This section provides the syntax of ALCH and offers an example of how it can be used. In addition, some categorical constructs, which will be used to define the semantics, are introduced. Section 3 gives a concise overview of relevant work both in the field of molecular and chemical computing and in the field of probabilistic semantics. The operational semantics for ALCH, and proofs that these semantics do indeed result in a sub-probability distribution, are given in Section 4. The denotational semantics are given in Section 5, along with examples that further aid the understanding of the more complex semantics in Section 5.6.

2 Background

Self-assembly is a process where particles form structures in a spontaneous manner [4]. An example of self-assembly that occurs naturally is the snowflake, a complex structure that is made-up of simple molecules. Self-assembling processes can be used in science to model biological systems, which are, like self-assembling systems, capable of self-organisation through molecules with specific purposes [2], but also for DNA-based computing [3]. In these fields, DNA molecules are deemed suitable particles for self-assembly, due to their rigid structure, the specific rules under which different DNA molecules can bind to each other and how well understood these rules are [2].
Work in this area is categorized as DNA nanotechnology, since it takes place at the molecular level. A distinction can be made between two classes of DNA nanotechnology. We are already familiar with the first class, which focuses on the self-assembly of structures through small DNA molecules. This class is called structural DNA nanotechnology. The second class models the behaviour of dynamic systems, such as for example logic systems, and is referred to as dynamic DNA nanotechnology [7]. Both classes have well-studied theoretical models, yet there has been little research on the interactions between these classes [2].

2.1 CRN-TAM

The Chemical Reaction Network-Controlled Tile Assembly Model (CRN-TAM) is a theoretical self-assembly model that studies the interaction of dynamic and structural DNA-based computing systems [2]. To achieve this, it combines elements from two other theoretical models, the abstract Tile Assembly Model and Chemical Reaction Networks. These will be explained below.

The abstract Tile Assembly Model (aTAM) is a Turing-universal framework that models DNA self-assembly in the 2D plane. The aTAM specifies rules by which an initial structure, called the seed assembly, can deterministically grow into a resultant structure given a set of tile types. Tiles, which are abstract representations of DNA molecules, are depicted as squares, where each side has a glue or binding strength, as well as a label. Tiles have a fixed orientation, meaning they cannot rotate [4].

Definition 1. A *tile type* is a four tuple $\boxed{t} = (N, E, S, W)$ with a north, south, east and west bond. Each bond is a tuple (l_B, s_B) with a bond label l_B and bond strength $s_B \in \mathbb{N}$. An example of a tile type is given in Figure 1.

The symbol T is used to denote the finite set of tile types, excluding the empty tile ϵ. Each element in T is a two tuple $(\boxed{t}, t^*)$, with t^* a signal species called the removal species. The removal species will be explained in Definition 3 and is not relevant until this definition.

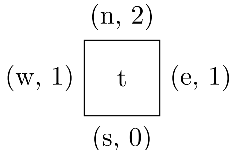


Figure 1: Example of a graphical representation of a tile type $\boxed{t}$, with label t and bonds N = (n, 2), E = (e, 1), S = (s, 0) and W = (w, 1).

A tile can *connect* to other tiles in an assembly. Two tile types can connect to each other if two of their opposite sides have the same bond label. The sides with which two tiles can connect to each other are called matching sides. If a tile is connected to at least one other tile, this tile is said to have a total bond strength. The total bond strength of a tile is the sum of the bond strengths of all sides that are connected to another tile. If two connected sides have different bond strengths, the smallest determines the strength of the connection[1]. An example of a tile connecting to an assembly is given below Definition 2, see Figure 2.

[1]This is the ALCH definition of total bond strength, given in [1]

Definition 2. An *assembly* is a function $A : \mathbb{Z}^2 \rightarrow (T \cup \{\epsilon\})$ that maps a coordinate to a tile type or the empty site ϵ. An aTAM model has a temperature $\tau \in \{0, 1, 2\}$ which determines which tiles can connect to and detach from an assembly (as seen in he following rules). An assembly must have the following properties:
- The so-called seed tile of the assembly, positioned at coordinate $(0, 0)$, cannot be empty. $A(0, 0) \neq \epsilon$.
- Each tile in the assembly must be indirectly connected to the seed tile. This means that at least one of each tiles sides must match with a tile next to it.
- The total binding strength of each tile in the assembly is at least τ.
An assembly can be referred to using a double box, like so: $\boxed{\boxed{x}}$. Its label is arbitrarily chosen and is independent of the tile types that make up the assembly.

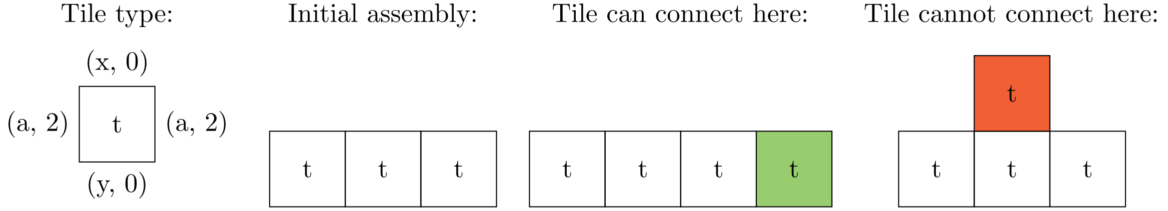


Figure 2: $\tau = 2$. Example of how a tile can connect to an assembly and how it cannot.

Self-assembly in aTAM is local, whether a tile can connect to an assembly at a certain coordinate depends on the tiles surrounding this coordinate (and the global temperature). Combining aTAM with the Chemical Reaction Network allows non-local control over tiles via chemical reactions.
Chemical Reaction Networks (CRN) formally model the behaviour of particles in a well-mixed solution by defining a set of chemical reactions [8]. These particles are usually referred to as signals, and the term signal species is used to refer to a type of signal. A CRN reaction is usually of the form $A + B \rightarrow C + D$, where A, B, C and D are signal species. Such a reaction can take place in a solution where there is at least one signal A and at least one signal B present. Mathematically, a reaction can be seen as an operation on a multiset of signal species. If the function $A + B \rightarrow C + D$ occurs, one element A and one element B are removed from the multiset, and one element C and one element D are added. Because CRN assumes a well-mixed solution, signals have no position or coordinate [2]. Including CRN reactions and signals in the CRN-TAM model allows the behaviour of tiles to be influenced on a non-local level.

Definition 3. A *CRN-TAM program* is a five tuple (Sp, T, R, τ, I), where:
- Sp is a finite set of signal species excluding the empty signal ϵ.
- T is a finite set of tile types.
- $\tau \in \{0, 1, 2\}$ is the (global) temperature.
- I is a finite multiset of tiles and signals that represents the initial state.
- R is a finite set of reactions of the following form, with signal species $A, B, C, D, t^*, r^*, x^* \in Sp \cup \{\epsilon\}$, tile types $(\boxed{t}, t^*), (\boxed{r}, r^*), (\boxed{x}, x^*) \in T$ and assemblies $\boxed{\boxed{x}}$ and $\boxed{\boxed{y}}$:

  - Signal reaction: $A + B \rightarrow^k C + D$
  - Tile deletion reaction: $A + \boxed{t} \rightarrow^k C + D$
  - Tile creation reaction: $A + B \rightarrow^k \boxed{t} + C$ or $A + B \rightarrow^k \boxed{t} + \boxed{r}$
  - Tile relabeling reaction: $A + \boxed{t} \rightarrow^k B + \boxed{r}$
  - Tile activation reaction: $A + \boxed{x} \rightarrow^k \boxed{\boxed{x}} + x^*$
  - Tile deactivation reaction: $\boxed{\boxed{x}} + x^* \rightarrow^k A + \boxed{x}$
  - Tile addition reaction: $\boxed{\boxed{x}} + \boxed{t} \rightarrow^k \boxed{\boxed{y}} + t^*$

- Tile removal reaction: $\boxed{\boxed{y}} + t^* \to^k \boxed{\boxed{x}} + \boxed{t}$

In these reactions k stands for the reaction rate. The reaction rate indicates the speed at which a reaction takes place.
A tile removal species is released every time a tile is activated or added to an assembly. This tile removal species needs to be present in the solution for the tile to be removed from the assembly or deactivated.

Note that tiles can be created and deleted by certain reactions. This means that whether a tile can connect to an assembly at a certain coordinate does not solely depend on the tiles surrounding this coordinate, but also on whether the tile exists in the solution at that moment in time.

2.2 ALCH

The Algorithmic Language for Chemistry (ALCH) is a high-level programming language for implementing CRN-TAM programs. ALCH was developed specifically to target CRN-TAM constructions that rely on sequences of reactions and tile attachments [1]. ALCH is not concurrent.
It is shown in [1] that ALCH can be used to strictly self-assemble the discrete Sierpinsky triangle and other infinite shapes. In order for a structure to be called strictly self-assembled, every coordinate that is not part of the structure must be empty. In contrast, weak self-assembly allows the use of "colourless" tiles (see Figure 3). aTAM, unlike ALCH, is only able to weakly self-assembly the discrete Sierpinsky triangle [9].

Figure 3: The outline of a square, weakly self-assembled on the left and strictly self-assembled on the right.

ALCH is an imperative language with a number of standard high-level features that supports three types of (global) variables. These types are bool, BondLabel and TileSpecies. The language includes CRN-TAM specific statements: activate, deactivate, add and remove, which take a TileSpecies as parameter. There is no parameter for the rate constant k which is present in CRN-TAM. This is probably because in ALCH only one reaction can happen at the same time due to the sequential nature of the language. At initialization of an ALCH program there are no assemblies present.
An interesting feature of ALCH is the branch statement. A branch consists of multiple paths that each have a boolean label and contain only add and remove statements. Paths are chosen at random until one finishes execution. If a path does not execute, the already executed add and remove statements are reversed before attempting another path. The same path can be attempted multiple times, even if it has already failed execution. This sounds inefficient, but makes sense once one remembers that reversing a path can result in a different assembly than the branch initially started with. The branch statement returns the boolean corresponding to the path that finished successfully. Paths can be given a weight which determines the probability that this path is chosen for an attempted execution. One could probably use the branch construct to emulate a rate constant, though that is not what it primarily designed for.

A formal ALCH grammar is not explicitly given in [1], though all relevant constructs are described. The following syntax was put together using the description in [1] and the ALCH compiler. We differentiate between commands C, boolean expressions *bexp* and reversible commands RC, which occur solely in branch

paths.
Some simplifications were introduced so that the syntax can function as a starting point for the semantics later on. Firstly, since there are no functions, procedures or scopes, we have removed variable declaration. The assumption is made that each variable exists outside any program. It is left to the user to assign a value to relevant variables before using a program. Because of this simplification, there are no BondLabel variables in this syntax anymore, since BondLabels are used exclusively to declare TileSpecies in ALCH. Secondly, if-then-else blocks are replaced by if-else blocks since these are less complex and functionally equivalent. In addition, a skip command was added. We define the composition of commands and reversible commands to be right-associative.

C ::= skip; | C_1 C_2 | var = $bexp$; | while ($bexp$) $\{C\}$ | if ($bexp$) $\{C_1\}$ else $\{C_2\}$

$bexp$ ::= v | b | !$bexp$ | $bexp_1$ && $bexp_2$ | $bexp_1$ || $bexp_2$ | ($bexp$)

Here, v is a variable and b a boolean value $b \in \mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$. Next up, we introduce the CRN-TAM specific commands and boolean expressions. We use $\boxed{t}$ to indicate a tile type. Branches are added to boolean expressions.

C ::= ... | activate $\boxed{t}$; | deactivate $\boxed{t}$; | add $\boxed{t}$; | remove $\boxed{t}$;

$bexp$ ::= ... | branch $\{L\}$

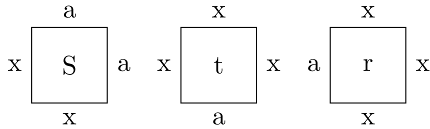
L is the finite weighted list of branch paths. Each branch path consists of reversible commands RC, a boolean value and a weight. ALCH includes unweighted branch expressions as well as weighted, but since an unweighted branch expression is equivalent to a branch expression in which each path has the same weight, the unweighted branch path is excluded from this (simplified) syntax.

$Path$ ::= false(i) $\{RC\}$ | true(i) $\{RC\}$ where $i \in \mathbb{I}$, and

RC ::= RC_1 RC_2 | add $\boxed{t}$; | remove $\boxed{t}$; | skip;

Example
We provide a simple example program, with global temperature $\tau = 2$, three tile types and two bond labels.

Temperature: $\tau = 2$
Bondlabels: $a = (a, 2)$, $x = (x, 1)$
Tiles:



ALCH code:

```
1    activate S;
2    v = branch {
3        true (2) { add t; }
4        false (1) { add r; }
5    }
6
7    while(v) {
8         v = branch {
9             true (1) { add r; }
```

```
10            false (1) { add S; }
11        }
12    }
```

Figure 4 shows the progress of the assembly through time. Starting from the empty assembly, the activation command (1) activates a tile of type $\boxed{S}$ with a probability of 1.0. The assignment command (2) contains a branch statement with two paths, which is why there are two edges from $\boxed{S}$, left corresponding to the `tt` path (3) and right corresponding to the `ff` path (4). Both of these path can execute. The last construct is a while loop (7), which is only entered when $v = \texttt{tt}$. Therefore the assembly in the left path is a terminal assembly. The while loop contains an assignment (8) with a branch statement. Here, the `tt` path (9) can execute, but the `ff` path (10) cannot, since the total bond strength with which $\boxed{S}$ can attach with is $1 < \tau = 2$, which is why it is reversed again. With $v = \texttt{tt}$, the while loop is entered again. Now, only the `ff` path can execute, which results in another terminal assembly.

Figure 4: The assembly throughout the example program

Another example program is provided in [1] (Figure 1).

2.3 Some Category Theory

An aspect of ALCH that will make specifying its semantics challenging is its stochastic nature. In this section, some mathematical constructs will be introduced that will be helpful in defining the semantics later on. In [6] the probabilistic semantics return a discrete sub-probability distributions over a set of program states S, defined as $DProb(S)$. $DProb(S)$ is the set of all functions $\omega : S \to [0, 1]$ such that $\sum_{s \in S} \omega(s) \leq 1$. Essentially, this construct is the same as the sub-distribution monad $\mathcal{D}$ [10], which is the notation that will be used throughout this thesis. To explain what this monad is and why it is useful to look at $DProb(S)$ as $\mathcal{D}(S)$ some categorical constructs need to be introduced. Only constructs relevant to this thesis will be discussed.

Definition 4. The *category of sets*, referred to as **Set**, consists of all sets, referred to as the objects of the category, and all functions between sets, which are called the morphisms of the category.

In defining the semantics a lot of sets will be involved, such as the set of program states and the set of boolean values $\{\texttt{tt}, \texttt{ff}\}$ for example. Therefore we can define all other categorical constructs using **Sets**.

Definition 5. A *functor* F is a mapping between categories that, given categories C and D, maps each object $X \in C$ to an object $F(X) \in D$ and maps each morphism $f : X \to Y \in C$ to a morphism $F(f) : F(X) \to F(Y) \in D$, s.t. $F(id_X) = id_{F(X)}$ and $F(g \circ f) = F(g) \circ F(f)$. An *endofunctor* is a functor which has the same domain and co-domain.

An example of a functor, and one that is relevant to this thesis, is an endofunctor that maps **Sets** to **Sets**. We cannot define the sub-distribution monad $\mathcal{D}$ with only this functor however. Another categorical construct is needed, that can map a functor to another functor.

Definition 6. A *natural transformation* maps one functor to another functor. Given two categories C and D, and two functors $F : C \to D$ and $G : C \to D$, a natural transformation associates to every $X \in C$ a morhpism $\eta_X : F(X) \to G(X)$, where $F(X), G(X) \in D$, in such a way that for every morphism $f : X \to Y \in C$, $\eta_Y \circ F(f) = G(f) \circ \eta_X$ holds (see Figure 5).

$$X \xrightarrow{f} Y \qquad \begin{array}{ccc} F(X) & \xrightarrow{\eta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\eta_Y]{} & G(Y) \end{array}$$

Figure 5: The composition of functors F and G and the natural transformation η.

Definition 7. A *monad* on **Sets** is a categorical construct that consists of:
- An endofunctor M : **Sets** $\to$ **Sets**.
- A natural transformation $\eta : I_{\mathbf{Sets}} \to M$, called the unit transformation, where $I_{\mathbf{Sets}}$ is the identity functor on the category **Set**. The unit transformation says that for any set X, there is a function $X \to^{\eta_X} M(X)$.
- A natural transformation $\mu : M^2 \to M$, called the multiplication transformation, where $M^2 = M \circ M$.
 The unit and multiplication transformation must satisfy (see Figure 6 also):
 - $\mu \circ M\mu = \mu \circ \mu M$
 - $\mu \circ M\eta = \mu \circ \eta M$.

$$\begin{array}{ccc} M(M(M(X))) & \xrightarrow{M(\mu_X)} & M(M(X)) \\ {\scriptstyle \mu_{M(X)}}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ M(M(X)) & \xrightarrow[\mu_X]{} & M(X) \end{array} \qquad \begin{array}{ccc} M(X) & \xrightarrow{\eta_{M(X)}} & M(M(X)) \\ {\scriptstyle M(\eta_X)}\downarrow & \searrow^{=} & \downarrow{\scriptstyle \mu_X} \\ M(M(X)) & \xrightarrow[\mu_X]{} & M(X) \end{array}$$

Figure 6: Commutativity diagrams illustrating the conditions that the natural transformations of a monad must satisfy.

In a branching structure, the multiplication μ from the monad allows the "flattening" of two branches into one [10]. A concrete example of this will be given once the sub-distribution monad $\mathcal{D}$ has been defined (see Figure 7).

Definition 8. The sub-distribution monad $\mathcal{D}$ on the endofunctor on **Sets** has the following natural transformations:

- The unit transformation $\eta^{\mathcal{D}}_X : X \to \mathcal{D}(X)$ is the *Dirac distribution*, $\eta^{\mathcal{D}}_X(x)(x') = \begin{cases} 1, & \text{if } x = x' \\ 0, & \text{otherwise.} \end{cases}$

- The multiplication transformation $\mu^{\mathcal{D}}_X : \mathcal{D}(\mathcal{D}(X)) \to \mathcal{D}(X)$ is defined as follows. Given a distribution D on distributions d on the set X, $\mu^{\mathcal{D}}_X(D)(x) = \sum_{d \in \mathcal{D}(x)} D(d) \cdot d(x)$

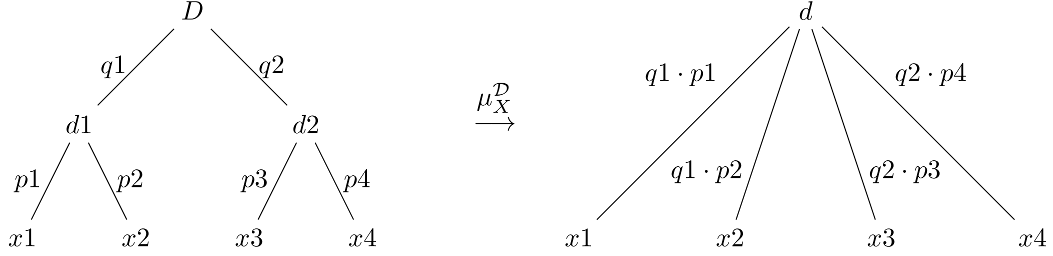


Figure 7: Illustration of the multiplication transformation $\mu^{\mathcal{D}}_X$, with $D : \mathcal{D}(\mathcal{D}(X))$, d, $d1$, $d2 : \mathcal{D}(X)$, $x1$, $x2$, $x3$, $x4 \in X$ and $q1$, $q2$, $p1$, $p2$, $p3$, $p4 \in [0,1]$.

The semantics that will be defined in this thesis will produce a sub-probability distribution over states and assemblies, meaning that they will be functions of the form $f : X \to \mathcal{D}(X)$. We cannot compose two functions $f : X \to \mathcal{D}(X)$ and $g : X \to \mathcal{D}(X)$. It will however be necessary in the definition of the semantics to compose such functions. There exists a category, called the Kleisli-category, whose composition operator does allow for the composition of f and g [10].

Definition 9. Given a monad M on **Sets**, the *Kleisli category* $\mathcal{K}\ell(M)$ is a category with sets as objects. Any morphism $X \to M(Y)$ in **Sets** is a morphism $X \to Y$ in $\mathcal{K}\ell(T)$. The *Kleisli composition* on two function $f : X \to M(Y)$ and $g : Y \to M(Z)$ is defined as $g \circledcirc^M f = \mu^M_z \circ T(g) \circ f$.

The Kleisli category on the sub-distribution monad $\mathcal{D}$ is be denoted by $\mathcal{K}\ell(\mathcal{D})$. Given two morphisms $f : X \to \mathcal{D}(X)$ and $g : X \to \mathcal{D}(X)$, its Kleisli composition $(g \circledcirc^{\mathcal{D}} f)$ calculates the sum $(g \circledcirc^{\mathcal{D}} f)(x)(x') = \sum_{x'' \in X} f(x)(x'') \cdot g(x'')(x')$ [10].

3 Related Work

There exist many variations and extensions on the Tile Assembly Model [4], of which CRN-TAM is not the only probabilistic variation. Another extension on aTAM is the Probabilistic Tile Assembly Model (PTAM), which, like CRN-TAM, assumes a well-mixed solution and allows multiple tiles to attach to an assembly in the same spot with a certain probability [11]. PTAM was developed specifically to model the assembly of one dimensional structures with the smallest possible tile type set, whereas CRN-TAM presents a two dimensional probabilistic aTAM variation that allows non-local control over tiles.

ALCH is not the first high-level programming language developed to implement programs based on solutions of molecules and chemical reactions. CRN++ is a high-level imperative language for specifying CRN programs, that uses constructs similar to standard constructs, like for example if-statements, to perform chemical reactions in real valued concentrations in a deterministic manner [12]. The so-called chemical metaphor has also proven to be useful in concurrent and distributed programming. In [13] the Chemical Abstract Machine (CHAM) is defined, a model of computer systems that is based on a multiset of molecules which interact according to certain rules. RCHAM is a low level extension of the CHAM, which makes the machine more suitable for distributed programming [14]. From RCHAM, the join-calculus was developed, which in turn resulted in a number of extensions on already existing high-level programming languages such as OCaml and Java [15]. The goal of these extensions is not to model molecular processes but to use the behaviour of chemical solutions as a way to execute concurrent computations.

In [6] (Chapter 2) the formal semantics for a simple imperative probabilistic programming language are given, starting with the classical constructs of the language before defining any semantics for the probabilistic part. This approach of defining semantics step by step served as direct inspiration for this thesis. The language in [6] is a quantum language, its probabilistic operations take place in the Hilbert space.

4 Operational Semantics

In this section the operational semantics for the language ALCH [1] will be defined, following the approach taken in Section 2 of [6]. Firstly the classical part of the syntax will be discussed, after which several CRN-TAM elements will be introduced in order of complexity. An overview of the final operational semantics is given at the end of this section.

4.1 Classical Constructs

ALCH includes a number of standard constructs for which the operational semantics are rather straightforward. We are concerned with commands C and boolean expressions *bexp*. In addition, a program state s is defined that allows the semantics to change and keep track of the values assigned to variables. Remember that there are only global variables, and that it is assumed that variable declaration happens before program execution.

Definition 10. A state s is a map $s : V \to \mathbb{B}$ with variables V and boolean values $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$. S is the finite set of all possible states s.

Definition 11. A *configuration* is a pair (C, s) with any command C and any state $s \in S$. A *boolean configuration* ia a pair $(bexp, s)$ with any boolean expression *bexp* and any state $s \in S$.

We define a reduction relation for boolean expressions $\hookrightarrow$ and a reduction relation for configurations $\to$.

Boolean expressions

$$(v, s) \hookrightarrow (s(v), s) \text{ (val)} \qquad (!b, s) \hookrightarrow (\neg\, b, s) \text{ (not)}$$

$$(b \mathbin{\&\&} b, s) \hookrightarrow (b \wedge b, s) \text{ (and)} \qquad (b \mid\mid b, s) \hookrightarrow (b \vee b, s) \text{ (or)}$$

$$\frac{(bexp, s) \hookrightarrow (bexp', s)}{(!bexp, s) \hookrightarrow (!bexp', s)} \text{ (notb)}$$

$$\frac{(bexp_1, s) \hookrightarrow (bexp_1', s)}{(bexp_1 \mathbin{\&\&} bexp_2, s) \hookrightarrow (bexp_1' \mathbin{\&\&} bexp_2, s)} \text{ (andb1)} \qquad \frac{(bexp, s) \hookrightarrow (bexp', s)}{(b \mathbin{\&\&} bexp, s) \hookrightarrow (b \mathbin{\&\&} bexp', s)} \text{ (andb2)}$$

$$\frac{(bexp_1, s) \hookrightarrow (bexp_1', s)}{(bexp_1 \mid\mid bexp_2, s) \hookrightarrow (bexp_1' \mid\mid bexp_2, s)} \text{ (orb1)} \qquad \frac{(bexp, s) \hookrightarrow (bexp', s)}{(b \mid\mid bexp, s) \hookrightarrow (b \mid\mid bexp', s)} \text{ (orb2)}$$

These inference rules are deterministic, the $\hookrightarrow$ relation is a partial function on pairs $(bexp, s)$ which is only undefined on (b, s), where b is a literal. One can see the pair (b, s) as the terminal boolean configuration.

Commands

$$(v = b;,\ s) \to (\text{skip};,\ s[v \mapsto b]) \text{ (assign)} \qquad \frac{(bexp, s) \hookrightarrow (bexp', s)}{(v = bexp;,\ s) \to (v = bexp';, s)} \text{ (assignb)}$$

$$(\text{skip};\ C,\ s) \to (C,\ s) \text{ (skip)} \qquad \frac{(C_1,\ s) \to (C_1',\ s')}{(C_1\ C_2,\ s) \to (C_1'\ C_2,\ s')} \text{ (comp)}$$

$$\frac{(bexp, s) \hookrightarrow (\texttt{tt}, s)}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s) \to (C_1,\ s)} \text{ (iftt)} \qquad \frac{(bexp, s) \hookrightarrow (\texttt{ff}, s)}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s) \to (C_2,\ s)} \text{ (ifff)}$$

$$\frac{(bexp, s) \hookrightarrow (bexp', s)}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s) \rightarrow (\text{if}\,(bexp')\,\{C_1\}\,\text{else}\,\{C_2\},\ s)}\ (\text{ifb})$$

$$\frac{(bexp, s) \hookrightarrow (\texttt{tt}, s)}{(\text{while}\,(bexp)\,\{C\},\ s) \rightarrow (C\ \text{while}\,(bexp)\,\{C\},\ s)}\ (\text{whilett}) \qquad \frac{(bexp, s) \hookrightarrow (\texttt{ff}, s)}{(\text{while}\,(bexp)\,\{C\},\ s) \rightarrow (\text{skip};,\ s)}\ (\text{whileff})$$

$$\frac{(bexp, s) \hookrightarrow (bexp', s)}{(\text{while}\,(bexp)\,\{C\},\ s) \rightarrow (\text{while}\,(bexp')\,\{C\},\ s)}\ (\text{whileb})$$

These inference rules are deterministic, the transition relation is a partial function on configurations which is undefined only on $(skip;, s)$. $(skip;, s)$ can be seen as the terminal configuration. We define the relation $\rightarrow^*$ to be the reflexive transitive closure of $\rightarrow$, and use it to define the operational semantics, a partial function $\mathcal{O}[\![C]\!] : S \rightharpoonup S$, as:

$$\mathcal{O}[\![C]\!](s) = \begin{cases} s' & \text{if } (C, s) \rightarrow^* (skip, s') \\ \texttt{undefined} & \text{if } (C, s) \text{ does not terminate.} \end{cases}$$

The while statement is the only command that may cause the operational semantics to be `undefined`, since it can loop indefinitely.

4.2 CRN-TAM Constructs

ALCH programs have a global temperature τ, the value of which cannot be altered in the middle of a program. This temperature can be accessed throughout the entire semantics. If one wanted to introduce a dynamic temperature (even though this is not possible in ALCH), one would have to alter the transition rules that will be defined in this section.
The set of assemblies in an ALCH program will be represented by a graph-like object G. This graph-like object G resembles an unconnected graph, which can represent multiple assemblies.

Definition 12. A graph G is a three tuple $\langle N, E, St\rangle$ of nodes N, edges E and seed tiles St, with:
- Nodes $N \subseteq T \times \mathbb{N}$, with T the finite set of tile types and $\mathbb{N}$ an integer id.
- Edges E a partial function $E : N \times \mathbb{D} \rightharpoonup N$, with $\mathbb{D} = \{N, E, S, W\}$ a direction.
- Seed tiles $St \subseteq N$.
$\mathbb{G}$ is the finite set of all possible states G.

Remember that the set of tile types T is the set of two tuples $(\boxed{t}, t^*)$, with a tile type $\boxed{t}$ and a removal species t^*. For ease of use, instead of this tuple we will often use $\boxed{t}$ instead, since the removal species it not relevant to our semantics or operations. For example, a node $n \in N$ will be written $(\boxed{t}, i)$ instead of $((\boxed{t}, t^*), i)$. Given a set the finite set of all possible tile types we can define $\mathcal{E} : T \times \mathbb{D} \times T$, the set of all matching tiles. A three tuple $(\boxed{t}, N, \boxed{r})$ is an element in $\mathcal{E}$ if the north bond of tile $\boxed{t}$ has the same bond label ans the south bond of tile $\boxed{r}$. We use $\boxed{t}_{d_S}$ to denote the bond strength of tile $\boxed{t}$ in direction d.

Each connected "subgraph" of G forms an assembly, where the seed of the assembly is the only node that is also present in the set of nodes St. A node is only added to St when it is activated. Tiles that are already connected, i.e. part of the domain or co-domain of E, cannot be activated. Two nodes $(\boxed{t}, i)$ and $(\boxed{r}, j)$ are directly connected if there exists a direction d for which either $E((\boxed{t}, i), d) = (\boxed{r}, j)$ or $E((\boxed{r}, j), d) = (\boxed{t}, i)$.
Every seed tile is given coordinates $(0, 0)$. The coordinates of a tile $(\boxed{t}, i) \notin St$ can be calculated using the only seed tile that it is connected to, by taking any path from that seed tile to itself. The x-coordinate then corresponds to (all east-pointing edges in the path - all west-pointing edges in the path), and is calculated

by the operation $x(\boxed{t}, i)$. The y-coordinate corresponds to (all north-pointing edges in the path - all south-pointing edges) in the path, and is calculated by the operation $y(\boxed{t}, i)$. In addition, we define an operation $con((\boxed{t}, i), (\boxed{r}, j))$ that returns $\top$ if the two given nodes are part of the same assembly, i.e. connected, and $\bot$ if they are not.

Graph Operations
To evaluate CRN-TAM commands, a number of operations need to be defined which can manipulate the graph-like object G. Firstly, an operation is required for tile activation, which adds a seed tile to a G with a unique id.

$$newNode(G, \boxed{t}, i) = \begin{cases} (\boxed{t}, i), & \text{if } (\boxed{t}, i) \notin N \\ newNode(\boxed{t}, i+1), & \text{otherwise} \end{cases}$$

► $activate\,(G, \boxed{t}) = \langle N \cup \{n\},\, E,\, St \cup \{n\} \rangle \qquad \text{where } n = newNode(G, \boxed{t}, 0)$

An operation is needed to deactivate an assembly. An assembly can only be deactivated if it consists of a single (seed) tile.

► $deactivate\,(G, \boxed{t}) = \{ \langle N \setminus \{(\boxed{t}, i)\},\, E,\, St \setminus \{(\boxed{t}, i)\} \rangle \mid \forall d.\ E((\boxed{t}, i), d) = \bot \}$

In addition, an operation is needed that removes a tile from an assembly. A tile can only be removed from an assembly if it is not the seed tile and if its total bond strength is τ exactly. The latter is checked using the *strength* operation. When a tile is removed from an assembly, its edges need to be removed as well, for its connection with the assembly is severed. This is handled by the *edgeRemove* operation.

$$strength(G, (\boxed{t}, i)) = \textstyle\sum_{(\boxed{r}, d, \boxed{q}) \in X} min(\boxed{r}_{d_S}, \boxed{q}_{d_S})$$
$$\text{where } X = \{\boxed{r}, d, \boxed{q} \mid \exists j, k.\, E((\boxed{r}, j), d) = (\boxed{q}, k) \land ((\boxed{r}, j) = (\boxed{t}, i) \lor (\boxed{q}, k) = (\boxed{t}, i))\}$$

$$edgeRemove(G, (\boxed{t}, i), (\boxed{r}, j), d) = \begin{cases} \bot, & \text{if } \boxed{r} = \boxed{t}, j = i \\ \bot, & \text{if } E((\boxed{r}, j), d) = (\boxed{t}, i) \\ E((\boxed{r}, j), d), & \text{otherwise} \end{cases}$$

► $remove\,(G, \boxed{t}) = \{ \langle N \setminus \{(\boxed{t}, i)\},\, edgeRemove(G, (\boxed{t}, i)),\, St \rangle \mid$
$(\boxed{t}, i) \in N,\ (\boxed{t}, i) \notin St),\ strength(G, (\boxed{t}, i)) = \tau \}$

Tiles should also be able to be added to assemblies. This is the most complicated operation, since a tile can only be added to an assembly if at least one of its sides match with a tile in the assembly and its total bond strength is greater than or equal to τ. The add operation has to find a tile in an assembly that can can connect with the new tile, check whether this coordinate is empty, check if the new tile has other sides that match with surrounding tiles and add all these edges to the graph-like object G. Firstly, the *add* operation finds all possible places where the new node can connect to another tile in an assembly in G, using the *possibleEdges* and *strength* operations. Once a tile has been connected to another tile, the *additionalEdges* operation adds any edges that this tile forms with adjacent tiles. In order to do that, it checks whether there are adjacent tiles and if they have matching sides using the *valid* operation.

$$empty(G, (\boxed{r}, j), d) = \begin{cases} N, & \text{if } d = N,\ con((\boxed{r}, j), (\boxed{q}, k)) = \top,\ x(G, (\boxed{q}, k)) \neq x(G, (\boxed{r}, j)), \\ & \quad y(G, (\boxed{q}, k)) \neq y(G, (\boxed{r}, r)) + 1 \\ E, & \text{if } d = E,\ con((\boxed{r}, j), (\boxed{q}, k)) = \top,\ x(G, (\boxed{q}, k)) \neq x(G, (\boxed{t}, j)) + 1, \\ & \quad y(G, (\boxed{q}, k)) \neq y(G, (\boxed{t}, j)) \\ S, & \text{if } d = S,\ con((\boxed{r}, j), (\boxed{q}, k)) = \top,\ x(G, (\boxed{q}, k)) \neq x(G, (\boxed{r}, j)), \\ & \quad y(G, (\boxed{q}, k)) \neq y(G, (\boxed{r}, j)) - 1 \\ W, & \text{if } d = W,\ con((\boxed{r}, j), (\boxed{q}, k)) = \top,\ x(G, (\boxed{q}, k)) \neq x(G, (\boxed{r}, j)) - 1, \\ & \quad y(G, (\boxed{q}, k)) \neq y(G, (\boxed{r}, j)) \\ \bot, & \text{otherwise} \end{cases}$$

$$coordinates(G, (\boxed{r}, j), d, (\boxed{t}, i)) = \begin{cases} N, & \text{if } d = N,\ x(G, (\boxed{r}, j)) = x(G, (\boxed{t}, i)), \\ & \quad y(G, (\boxed{r}, j)) = y(G, (\boxed{t}, i)) - 1 \\ E, & \text{if } d = E,\ x(G, (\boxed{r}, j)) = x(G, (\boxed{t}, i)) - 1, \\ & \quad y(G, (\boxed{r}, j)) = y(G, (\boxed{t}, i)) \\ S, & \text{if } d = S,\ x(G, (\boxed{r}, j)) = x(G, (\boxed{t}, i)), \\ & \quad y(G, (\boxed{r}, j)) = y(G, (\boxed{t}, i)) + 1 \\ W, & \text{if } d = W,\ x(G, (\boxed{r}, j)) = x(G, (\boxed{t}, i)) + 1, \\ & \quad y(G, (\boxed{r}, j)) = y(G, (\boxed{t}, i)) \\ \bot, & \text{otherwise} \end{cases}$$

$$\begin{aligned} & valid(G, (\boxed{t}, i)) = \{((\boxed{r}, j), d) \mid \\ & \quad (\boxed{r}, d, \boxed{t}) \in \mathcal{E} \wedge \\ & \quad con((\boxed{t}, i), (\boxed{r}, j)) = \top \wedge \\ & \quad coordinates(G, (\boxed{r}, j), d, (\boxed{t}, i)) = d\} \end{aligned}$$

$$addEdges(G, (\boxed{t}, i), (\boxed{r}, j), d) = \begin{cases} (\boxed{t}, i), & \text{if } ((\boxed{r}, j), d) \in valid(G, (\boxed{t}, i)) \\ E((\boxed{r}, j), d), & \text{otherwise} \end{cases}$$

$$additionalEdges(G, (\boxed{t}, i)) = E[\,((\boxed{r}, j), d) \mapsto addEdges(G, (\boxed{t}, i), (\boxed{r}, j), d)\,]$$

$$possibleEdges(G, (\boxed{t}, i)) = \{E[\,((\boxed{r}, j), d) \mapsto (\boxed{t}, i)\,] \ \mid \ (\boxed{r}, d, \boxed{t}) \in \mathcal{E} \wedge empty(G, (\boxed{r}, j), d)) = d\,\}$$

► $add\,(G, \boxed{t}\,) = \{\langle N', additionalEdges(\langle N', E', St\rangle, n), St\rangle \mid$
$E' \in possibleEdges(G, n) \wedge strength(\langle N', additionalEdges(\langle N', E', St\rangle, n), St\rangle, n) \geq \tau\}$
where $n = newNode(G, \boxed{t}, 0)$ and $N' = N \cup \{n\}$

Small-step Operational Semantics

The graph-like object $G \in \mathbb{G}$ is added to the configuration, so that it becomes (C, s, G). This configuration is used to define the following inference rules. These rules are not deterministic, therefore each transition relation is labeled with a probability p such that, given any configuration, for all transitions $(C, s, G) \rightarrow^p (C', s', G')$ the sum of all p is at most one. The symbol γ is used to denote a pair (s, G) when that improves readability, and $\Gamma : S \times \mathbb{G}$ is used to denote the finite set of all (s, G).

$$(\text{activate } \boxed{t};\ , s,\ G) \rightarrow^1 (\text{skip};\, , s, activate(G, \boxed{t})) \ (\text{act})$$

$$\frac{\mathcal{G} = deactivate(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{deactivate } \boxed{t};\ , s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip};\ , s,\ \mathcal{G}_i)} \text{ (deact)}$$

$$\frac{\mathcal{G} = add(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{add } \boxed{t};\ , s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip};\ , s,\ \mathcal{G}_i)} \text{ (add)} \qquad \frac{\mathcal{G} = remove(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{remove } \boxed{t};\ , s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip}, s,\ \mathcal{G}_i)} \text{ (rem)}$$

Note that for any configuration with a CRN-TAM command, the sum of all probabilities corresponding to single reductions from this configuration is $p \leq 1$, since the relation $\rightarrow^p$ is not left-total. There is no transition for CRN-TAM commands for which the corresponding $\mathcal{G}$ is empty.
The previously defined transitions are also given a probability label and a configuration which includes the graph-like object G. This G is altered only in the composition rule. Every $\rightarrow$ is labeled with a probability of 1, except in the composition rule. For composition, both the probability p and graph G' in the premise are passed on to the conclusion. See Section 4.4 for the final version of the inference rules.

Definition 13. A *computation* $Comp(c, \gamma, \gamma')$ for any given C, γ and γ' is the set of all sequences from initial configuration (C, γ) to the terminal configuration $(skip;, \gamma')$, like so:

$$(C, \gamma) = (C_0, \gamma_0) \rightarrow^{p1} \ldots \rightarrow^{pk} (C_k, \gamma_k) = (skip;, \gamma').$$

For any sequence c, the total probability $p(c)$ is the product $\Pi_{i=1}^{k}\ p_i$.

Definition 14. The *operational sub-probability distribution* is the total function $\mathcal{O}[\![C]\!] : \Gamma \rightarrow \mathcal{D}(\Gamma)$, where $\mathcal{D}$ is the sub-distribution monad. $\mathcal{D}$ can be seen as the set of discrete sub-probability distributions on Γ, i.e.: functions $\omega : \Gamma \rightarrow [0, 1]$ such that $\sum_{\gamma \in \Gamma} \omega(\gamma) \leq 1$. Given any command C we define $\mathcal{O}[\![C]\!]$ as follows:

$$\mathcal{O}[\![C]\!](\gamma)(\gamma') = \sum \{p(c) \mid c \in Comp(C, \gamma, \gamma')\}.$$

Lemma 1. For any command C and states γ and γ', $\mathcal{O}[\![C]\!](\gamma)(\gamma') \leq 1$.

Proof. We will prove this by induction on the tree representation of all $c \in Comp(C, \gamma, \gamma')$. This tree has root node (C, γ), from which all paths follow the transitions of a sequence in $Comp(C, \gamma, \gamma')$. All leaves in this tree are therefore $(skip;, \gamma')$. The edges are labeled with a probability p. This tree has finite depth, and will be referred to as the reduction tree. An example of a reduction tree is given in Figure 8

Base case A reduction tree of depth $d = 1$ represents a single sequence c that consist of only one transition. This tree cannot represent multiple sequences since no transition can reduce to the a configuration $(\text{skip};, \gamma')$ from a initial configuration (C, γ) in different ways. The transitions that can appear in this tree are those that reduce to a terminal configuration in one step. These transitions are: (assign), (whileff), (act), (deact), (add) and (remove). The first three transitions reduce with a probability $p = 1$. The latter three reduce with a probability $p = 1/|\mathcal{G}| \leq 1$, since $|\mathcal{G}| > 0$. Therefore $\mathcal{O}[\![C]\!](\gamma)(\gamma') \leq 1$ for any tree of depth 1.

Induction step We assume a reduction tree of depth $d > 1$ corresponding to all c in a $Comp(C, \gamma, \gamma')$, with $k > 0$ outgoing paths from the root node (C, γ), which all end in a leaf node (C', γ') with $C' = \text{skip};$. Each of these paths is itself a tree, from node (C'', γ^i) to one or more leaf nodes $(skip;, \gamma')$, with $0 < i \leq k$. We assume an induction hypothesis $\forall\ 0 < i \leq k.\ \mathcal{O}[\![C'']\!](\gamma^i)(\gamma') \leq 1$ and set out to prove $\mathcal{O}[\![C]\!](\gamma)(\gamma') \leq 1$.
We know that the initial command C can not reduce to a terminal state in a single step, since $d > 1$ means that it cannot reduce to a leaf. This means that the root node of the reduction tree can be one of the following transitions: (assignb), (skip), (comp), (iftt), (ifff), (ifb), (whilett), (whileb).

Case (assignb), (skip), (iftt), (ifff), (ifb), (whilett), (whileb)
These transitions are deterministic and have a corresponding probability $p = 1$. Therefore, the amount

of subtrees k following from this transition can only be $k = 1$. By definition of $\mathcal{O}$, $\mathcal{O}[\![C]\!](\gamma)(\gamma') = p * \mathcal{O}[\![C'']\!](\gamma^1)(\gamma') = 1 * \mathcal{O}[\![C'']\!](\gamma^1)(\gamma') \leq 1$ by the induction hypothesis.

Case (comp)

It is worth noting that every transition has its own derivation tree, of which the leaves are transitions without any further transitions as premise. An example of some derivation trees is given in Figure 9. Note the difference between the reduction tree and derivation trees throughout the rest of the proof.
Due to the nature of small step semantics and the defined inference rules, each (comp) derivation tree, or any other derivation tree corresponding to a C transition, has only one leaf. This is obvious since every transition has only one or no C transitions in its premise. The derivation tree of a (comp) transition can have any transition but (comp) as its leaf, since only (comp) has another transition in its premise. The only no-leaf nodes in a (comp) derivation tree are therefore other (comp) transitions. Since the probability p of a (comp) transition is equal to the probability of the transition in its premise, the probability of any (comp) transition is the probability of the leaf of its derivation tree. If this leaf is deterministic, the entire derivation tree is deterministic and has probability $p = 1$. This means that there is only one possible reduction subtree that it reduces to, meaning that by definition of $\mathcal{O}$, $\mathcal{O}[\![C_1 C_2]\!](\gamma)(\gamma') = p * \mathcal{O}[\![C'']\!](\gamma^0)(\gamma') = 1 * \mathcal{O}[\![C'']\!](\gamma^0)(\gamma') \leq 1$ by the induction hypothesis. This is the case for leaf-transitions (assign), (assignb), (skip), (iftt), (ifff), (iffb), (whilett), (whilett), (whileb) and (act).

If the leaf of the (comp) derivation tree is a (deact), (add) or (rem) transition, the probability of the (comp) transition is $p = 1/|\mathcal{G}|$, with $\mathcal{G} > 0$ by definition of the inference rules. The amount of different reduction subtrees (that transition to (skip; γ')) that this transition can reduce to is $k \leq |\mathcal{G}|$. By definition of $\mathcal{O}$, $\mathcal{O}[\![C_1\ C_2]\!] = \sum_{i=0}^{k} \frac{1}{|\mathcal{G}|} * \mathcal{O}[\![C'']\!](\gamma^i)(\gamma')$. By the induction hypothesis we know that any $\mathcal{O}[\![C'']\!](\gamma^i)(\gamma') \leq 1$ with $0 \leq i < k$. By the definition of k we know that $k * \frac{1}{|\mathcal{G}|} \leq 1$, therefore $\mathcal{O}[\![C]\!](\gamma)(\gamma') \leq 1$. □

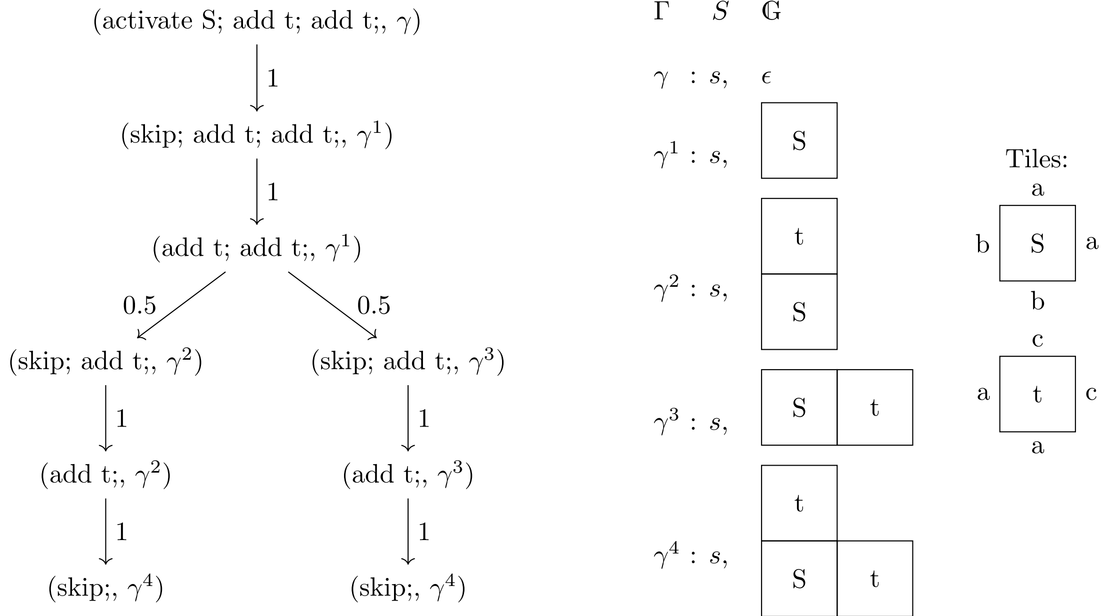


Figure 8: Example of a reduction tree ($\tau = 0$, bond strengths are irrelevant).

tree 1:
$$\frac{(\text{activate } S;,\ G,\ s) \rightarrow^1 (\text{skip};,\ \mathit{activate}(G,S),\ s)}{(\text{activate } S;\ \text{add } t;\ \text{add } t;, s, G) \rightarrow^1 (\text{skip};\ \text{add } t;\ \text{add } t;,\ s,\ G')}$$

tree 2:
$$(\text{skip};\ \text{add } t;\ \text{add } t;,\ s,\ G') \rightarrow^1 (\text{add } t;\ \text{add } t;,\ s,\ G')$$

tree 3:
$$\frac{\dfrac{\mathcal{G} = \mathit{add}(G,t) \quad \mathcal{G} \neq \{\} \quad 0 \leq i < |\mathcal{G}|}{(\text{add } t;,\ s,\ G') \rightarrow^{0.5} (\text{skip};,\ s,\ \mathcal{G}_{i=0})}}{(\text{add } t;\ \text{add } t;,\ s,\ G') \rightarrow^{0.5} (\text{skip};\ \text{add } t;,\ s,\ \mathcal{G}_0)}$$

Figure 9: The derivation trees corresponding to the first three transitions in the reduction tree.

4.3 The Branch Construct

This section the semantics for the boolean branch expression is defined. This is a probabilistic expression, therefore the boolean reduction relation $\hookrightarrow$ is labeled with a probability p.

4.3.1 Branch Paths and the Backtracking Relation

Before the operational semantics of the branch expression can be defined, a backtracking configuration needs to be defined. This will allow us to backtrack any branch path that was unable to finish executing. It is not possible to simply take the state from before the (attempted) execution of the branch path due, to the non-deterministic nature of the add and remove constructs. For example, if a tile of type $\boxed{t}$ is added to an assembly with other tiles of this type already present, reversing this command may remove any tile of type $\boxed{t}$. In addition, it is possible to add a tile with a total bond strength greater than τ, which makes it impossible to remove, since a tile can only be removed if its total bond strength is equal to τ.

Definition 15. A *backtracking configuration* is a three tuple (RC, G, c), with RC a reversible command, G the graph-like object and c a finite list of commands that can be used to backtrack RC when it does not terminate.

We will use $\Rightarrow_b$ as the backtracking relation. The backtracking relation reduces *add* and *remove* commands, which are not deterministic, and is therefore also labeled with a probability. The element $\perp$ is introduced to indicate that a path is unable to execute completely, and thus must be backtracked. The co-domain of the relation $\Rightarrow_b$ is the union of the set of tuples (G, c) and the set $\{\perp, \epsilon\}$, where ϵ is the empty element. Note that whenever ϵ occurs in the semantics, it is simply not written. We give the big-step operational semantics for RC as a set of inference rules.

$$(\mathit{skip};, G, c) \Rightarrow_b^1 (G, c) \text{ (RCskip)}$$

$$\frac{\mathcal{G} = \mathit{add}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{add } \boxed{t};, G, c) \Rightarrow_b^{1/|\mathcal{G}|} (\mathcal{G}_i,\ \text{remove } \boxed{t}; +c)} \text{ (RCadd)} \qquad \frac{\mathit{add}(G, \boxed{t}) = \{\}}{(\text{add } \boxed{t};, G, c) \Rightarrow_b^1 (\perp, G, c)} \text{ (RCadd}_\perp)$$

$$\frac{\mathcal{G} = \mathit{remove}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{remove } \boxed{t};, G, c) \Rightarrow_b^{1/|\mathcal{G}|} (\mathcal{G}_i,\ \text{add } \boxed{t}; +c)} \text{ (RCrem)} \qquad \frac{\mathit{remove}(G, \boxed{t}) = \{\}}{(\text{remove } \boxed{t};, G, c) \Rightarrow_b^1 (\perp, G, c)} \text{ (RCrem}_\perp)$$

$$\frac{(RC_1, G, c) \Rightarrow_b^{p_1} (G'', c'') \qquad (RC_2, G'', c'') \Rightarrow_b^{p_2} (G', c')}{(RC_1\ RC_2, G, c) \Rightarrow_b^{p_1 * p_2} (G', c')} \text{ (RCcomp)}$$

$$\frac{(RC_1, G, c) \Rightarrow_b^{p} (\perp, G', c')}{(RC_1\ RC_2, G, c) \Rightarrow_b^{p} (\perp, G', c')} \text{ (RC}\perp 1)$$

$$\frac{(RC_1, G, c) \Rightarrow_b^{p_1} (G'', c'') \qquad (RC_2, G'', c'') \Rightarrow_b^{p_2} (\bot, G', c')}{(RC_1\ RC_2, G, c) \Rightarrow_b^{p_1 * p_2} (\bot, G', c')} \text{ (RC}\bot\text{2)}$$

The composition $RC_1\ RC_2$ is right associative. Note that the backtracking relation is left-total.

Definition 16. The discrete *operational backtrack distribution* is defined by the following operators:

$$\mathcal{O}_{\mathcal{RC}}[\![RC]\!](G, c)(G', c') = \sum\{p_i \mid (RC, G, c) \Rightarrow_b^{p_i} (G', c')\}$$

and

$$\mathcal{O}_{\mathcal{RC}\bot}[\![RC]\!](G, c)(\bot, G', c') = \sum\{p_i \mid (RC, G, c) \Rightarrow_b^{p_i} (\bot, G', c')\}.$$

Lemma 2. For any configuration (G', c'), given an initial (RC, G, c), $\mathcal{O}_{\mathcal{RC}}[\![RC]\!](G, c)(G', c') \leq 1$.

Proof. We will prove this by case analysis on RC. The reversible commands that reduce to (G', c') are (RC-skip), (RCadd), (RCrem) and (RCcomp).

Case (RCskip)
(RCskip) is a deterministic transition with $p = 1 \leq 1$.

Case (RCadd)
All $G \in \mathcal{G}$ in the premise of (RCadd) are unique by the definition of set. Therefore, the sum $\sum\{p_i \mid (add\ \boxed{t};, G, c) \Rightarrow_b^{p_i} (G', c')\}$ contains at most one element. The probability p of (RCadd) is $p = 1/|G| \leq 1$, with $|G| > 0$ by definition. Therefore, $\mathcal{O}_{\mathcal{RC}}[\![add\ \boxed{t};]\!](G, c)(G', c') \leq 1$.

Case (RCrem)
The same reasoning applies here as (RCadd).

Case (RCcomp)
The probability corresponding to a (RCcomp) transition is the product of the probabilities of all the leaves in the corresponding derivation tree. The leaves of an (RCcomp) derivation tree are always (RCskip), (RCadd) and/or (RCrem), and any other nodes are (RCcomp) transitions. We take n the amount of leaves and $|\mathcal{G}^i|$ the amount of transitions possible from the ith leave, with $0 < i \leq n$. The probability of a (RCcomp) transition is the product of the probabilities of the leaves, making $p = \frac{1}{|\mathcal{G}^1|} \times \cdots \times \frac{1}{|\mathcal{G}^n|} = \frac{1}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|}$. The maximum amount of different derivation trees to a (G', c) from the same (RCcomp) is at most $\mathcal{G}^1 \times \ldots \times \mathcal{G}^n$. Therefore the sum of probabilities $\mathcal{O}_{\mathcal{RC}}[\![RC_1 RC_2]\!](G, c)(G', c') \leq \frac{1}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|} \times \mathcal{G}^1 \times \ldots \times \mathcal{G}^n = \frac{\mathcal{G}^1 \times \ldots \times \mathcal{G}^n}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|} = 1$ □

Lemma 3. For any configuration (G', c'), given an initial (RC, G, c), $\mathcal{O}_{\mathcal{RC}\bot}[\![RC]\!](G, c)(\bot, G', c') \leq 1$.

Proof. We will prove this by case analysis on RC. The reversible commands that reduce to $(\bot, G', c')$ are (RCadd$_\bot$), (RCrem$_\bot$), (RC$\bot$1) and (RC$\bot$2).

Case (RCadd$_\bot$)
This is a deterministic transition with no further transitions in its premise. The sum $\sum\{p_i \mid (add\ \boxed{t};, G, c) \Rightarrow_b^{p_i} (\bot, G', c')\}$ contains at most one element. The probability p of (RCadd$_\bot$) is $p = 1$, therefore $\mathcal{O}[\![add\ \boxed{t};]\!](G, c)(\bot, G', c') \leq 1$.

Case (RCrem$_\bot$)
The same reasoning applies here as case (RCadd$_\bot$)

Case (RC$\bot$1), (RC$\bot$2)
The derivation trees of both (RC$\bot$1) and (RC$\bot$2) contain at least one (RCadd$_\bot$) or (RCrem$_\bot$) leaf, and

may also contain (RCskip), (RCadd) and (RCrem) leaves. The derivation trees may contain (RCcomp), (RC⊥1) and (RC⊥2) nodes. The probability of the entire reduction is the product of all the probabilities of the leaves. If there are no (RCadd) or (RCrem) leaves, this means that the reduction is deterministic, meaning that there is only one possible derivation tree. Therefore in this case the probability is $\mathcal{O}_{\mathcal{RC}}[\![RC_1 RC_2]\!](G,c)(\bot,G',c') = 1 \leq 1$.
If there are (RCadd) and/or (RCrem) leaves, the probability of the reduction tree is the product $\frac{1}{|\mathcal{G}^1|} \times \cdots \times \frac{1}{|\mathcal{G}^n|} = \frac{1}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|}$, with n the amount of (RCadd) and (RCrem) leaves and $|\mathcal{G}^i|$ the amount of transitions possible from the ith leaf (with $0 \leq i < n$). The maximum amount of derivation trees to a $(\bot, G', c')$ outcome from a (RC⊥1) or (RC⊥2) transition is at most $\mathcal{G}^1 \times \ldots \times \mathcal{G}^n$. Therefore the sum of probabilities is $\mathcal{O}_{\mathcal{RC}}[\![RC_1 RC_2]\!](G,c)(\bot,G',c') \leq \frac{1}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|} \times \mathcal{G}^1 \times \ldots \times \mathcal{G}^n = \frac{\mathcal{G}^1 \times \ldots \times \mathcal{G}^n}{|\mathcal{G}^1| \times \cdots \times |\mathcal{G}^n|} = 1$ □

4.3.2 Semantics of the Branch Expression

We define $|L|$ the size of the list L of branch paths. We define L_i the ith branch path of the list, which is essentially a two tuple of a boolean value and reversible commands. We define the operation $bool(L_i)$ that returns the boolean value of the branch path and the operation $body(L_i)$ that returns the reversible commands RC of the given branch path. As discussed in the previous section, the backtracking relation $\Rightarrow_b^p$ returns a list c, that contains the commands necessary to backtrack the initial reversible commands, and a graph G'. The commands in c need to be executed when the execution of a path fails. We define the function $w(L)$ that returns the sum of the weights of all branch paths in the given list L.

$$\frac{0 \leq i < |L| \qquad (body(L_i), G,\ \text{skip};) \Rightarrow_b^p (G', c') \qquad b = bool(L_i)}{(\text{branch}\{L\}, s, G) \hookrightarrow^{w(L_i)/w(L) * p} (b, s, G')} \text{(branch)}$$

$$\frac{0 \leq i < |L| \qquad (body(L_i), G,\ \text{skip};) \Rightarrow_b^{p_1} (\bot, G'', c) \qquad (c, G'',\ \text{skip};) \Rightarrow_b^{p_2} (G', c')}{(\text{branch}\{L\}, s, G) \hookrightarrow^{w(L_i)/w(L) * p_1 * p_2} (\text{branch}\{L\}, s, G')} \text{(branch}_\bot\text{)}$$

Observe that the $\hookrightarrow^p$ relation is not left-total. There is no inference rule for the situation where a configuration (c, G'', ϵ) reduces to a $(\bot, G', c')$. We do not want to attempt to reverse the already reversed, and thus failed, commands. Also note that a branch expression can loop forever when there is no branch path that can be executed completely.
With the introduction of probabilistic boolean expressions, that can add and remove tiles from a given graph, a single *bexp* command may, from a given γ, lead to the a terminal (b, γ') in multiple transition sequences. Remember that γ is shorthand for the pair (s, G).

Definition 17. A *Boolean computation* $Comp_B(bexp, \gamma, (b, \gamma'))$ for any given *bexp*, γ, b and γ' is the set of all sequences from initial boolean configuration $(bexp, \gamma)$ to terminal boolean configuration (b, γ'), like so:

$$(bexp, \gamma) = (bexp_0, \gamma_0) \hookrightarrow^{p_1} \ldots \hookrightarrow^{p_k} (bexp_k, \gamma_k) = (b, \gamma').$$

For any sequence c_B we define $p(c_B)$ the product $\Pi_{i=1}^k\ p_i$.

Definition 18. The *operational sub-probability distribution for boolean expressions* is the total function $\mathcal{O}_{\mathcal{B}}[\![bexp]\!] : \Gamma \to \mathcal{D}(\mathbb{B} \times \Gamma)$, where $\mathcal{D}$ is the sub-distribution monad. $\mathcal{D}(\mathbb{B} \times \Gamma)$ can be seen as the set of discrete sub-probability distributions on $(\mathbb{B} \times \Gamma)$, i.e.: functions $\omega_B : (\mathbb{B} \times \Gamma) \to [0,1]$ such that $\sum_{(b,\gamma) \in \mathbb{B} \times \Gamma} \omega_B(b, \gamma) \leq 1$. Given any boolean expression *bexp* we define $\mathcal{O}_{\mathcal{B}}[\![bexp]\!]$ as follows:

$$\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b, \gamma') = \sum \{p(c) \mid c \in Comp_B(bexp, \gamma, (b, \gamma'))\}$$

The previously defined inference rules need to be adapted now that the relation $\hookrightarrow$ has a probability label p and returns a graph-like object G' as well as a boolean literal and a state s. The transitions that have no premise and only literals in their conclusion are simply given a probability label of 1, and do not change the given G. These transitions are (val), (not), (and) and (or). The other transitions, (notb), (andb1), (andb2),

(orb1) and (orb2) all further the evaluation of a $bexp_i$ to a $bexp'_i$. The probability label given to such a reduction is the same probability with which this $bexp_i$ reduces to $bexp'_i$ in the premise of the transition. The outcome graph G^i in the premise is also the outcome graph in the conclusion of the inference rule. The final *bexp* inference rules are given in Section 4.4.

Lemma 4. $\mathcal{O}_{\mathcal{B}}[\![branch(L)]\!](\gamma)(b,\gamma')$ implies the following inference rules:

$$\frac{0 \leq i < |L| \qquad p = \mathcal{O}_{\mathcal{RC}}[\![body(L_i)]\!](G,\epsilon)(G',c') > 0 \qquad b = bool(L_i)}{(\text{branch}\{L\}, s, G) \hookrightarrow^{w(L_i)/w(L)\,*\,p} (b, s, G')} \text{ (_branch)}$$

$$\frac{\begin{array}{c} 0 \leq i < |L| \qquad p_1 = \mathcal{O}_{\mathcal{RC}\perp}[\![body(L_i)]\!](G,\epsilon)(\perp, G'', c) > 0 \\ p_2 = \mathcal{O}_{\mathcal{RC}}[\![c]\!](G'',\epsilon)(G',c') > 0 \end{array}}{(\text{branch}\{L\}, s, G) \hookrightarrow^{w(L_i)/w(L)\,*\,p_1\,*\,p_2} (\text{branch}\{L\}, s, G')} \text{ (_branch}_\perp)$$

Proof. $\mathcal{O}_{\mathcal{B}}[\![branch(L)]\!](\gamma)(b,\gamma')$ is the sum of the probabilities of all reduction sequences from $(branch(L),\gamma)$ to (b,γ'). Each single reduction step of a branch construct has one or two reductions with the backtrack relation $\Rightarrow_b^p$ in its premise and no other reductions. All backtrack reductions in premises of transitions in the sequences from $(branch(L),\gamma)$ to (b,γ') are included in $\mathcal{O}_{\mathcal{B}}[\![branch(L)]\!](\gamma)(b,\gamma')$. This implies (_branch) and (_branch$_\perp$). □

Lemma 5. For any *bexp*, initial state γ and pair (b,γ'), $\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b,\gamma') \leq 1$.

Proof. We will prove this by induction on the tree representation of all $c \in Comp_B(bexp,\gamma,(b,\gamma'))$. This tree has root node $(bexp,\gamma)$, from which all paths follow the transitions of a sequence in $Comp_B(bexp,\gamma,(b,\gamma'))$. All leaves in this tree are therefore (b,γ'). The edges are labeled with a probability p. This tree will be referred to as the reduction tree, and may have a countable infinite depth due to the (branch$_\perp$) transition, which may loop infinitely until a right path is chosen. Any reduction tree without this transition has finite depth.

Base case A reduction tree of depth 1 represents a single sequence c that consists of only one transition. The only valid transitions are (val), (not), (and), (or) and (branch), since these transitions lead to a literal in one reduction step.
The first four transitions reduce with probability $p = 1$. The last reduces with probability $p = w(L_i)/w(L) \leq 1$ with $0 \leq i < |L|$. Therefore $\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b,\gamma') \leq 1$ for any reduction tree of depth 1.

Induction step We assume a reduction tree of depth $d > 1$ corresponding to all c in a $Comp_B(bexp,\gamma,(b,\gamma'))$, with $k > 0$ outgoing paths from the root node $(bexp,\gamma)$, which all end in a leaf node (b,γ'). Each of these paths is itself a reduction tree, from node $(bexp',\gamma^i)$ to (b,γ'), with $0 \leq i < k$. We assume an induction hypothesis $\forall\ 0 \leq i < k.\ \mathcal{O}_{\mathcal{B}}[\![bexp']\!](\gamma^i)(b,\gamma') \leq 1$ and set out to prove $\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b,\gamma') \leq 1$.
We know *bexp* does not reduce to a terminal state, since it is not a leaf in the reduction tree. This means that *bexp* can be one of the following transitions: (notb), (andb1), (andb2), (orb1), (orb2) and (branch⊥).

Case (notb), (andb1), (andb2), (orb1), (orb2)
Every transition has its own derivation tree, of which the leaves are transitions without any further transitions in their premise. Due to the nature of small step semantics and the defined inference rules, each derivation tree has only one leaf. This is clear from the given that any transition has only one or no *bexp* transitions in its premise. Possible leaves in such a derivation tree are (val), (not), (and), (or), (branch) and (branch⊥). The first four of these are deterministic with $p = 1$. This means that the probability of the entire reduction is also 1, since all transitions with a transition in its premise have the same probability as the probability in the premise. Simply put, in any such derivation tree, the probability is always passed on from the leaf to the root node. Since the tree is deterministic, it can only reduce to $k = 1$ subtrees. By definition of $\mathcal{O}_{\mathcal{B}}$, $\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b,\gamma') = p * \mathcal{O}_{\mathcal{B}}[\![bexp']\!](\gamma^0)(\gamma') = 1 * \mathcal{O}_{\mathcal{B}}[\![bexp']\!](\gamma^0)(\gamma') \leq 1$ by the induction

hypothesis.

If the leaf of such a derivation tree is the (branch) or ($\text{branch}_\perp$) transition, it is not deterministic. Lemma 4 allows us to substitute (_branch) for (branch) and ($\text{_branch}_\perp$) for ($\text{branch}_\perp$). We know from proposition 2 that $\mathcal{O}_{\mathcal{RC}}[\![RC]\!](G,c)(G',c') \leq 1$ and $\mathcal{O}_{\mathcal{RC}\perp}[\![RC]\!](G,c)(\perp,G',c') \leq 1$ for any RC, (G,c), (G',c') and $(\perp,G',c')$. In addition, we know that the backtrack relation is left-total and that $\mathcal{O}_{\mathcal{RC}}[\![RC]\!](G,c)(G',c')$ and $\mathcal{O}_{\mathcal{RC}\perp}[\![RC]\!](G,c)(\perp,G',c')$ represent a discrete probability distribution by definition. The probability of the transitions does not only depend on the execution of its body, but also on the weight of the path and the sum of the weights of all paths. We know, by definition of the inference rules, that $\sum_{i=0}^{|L|} w(L_i)/w(L) = 1$. The sum of probabilities $\sum p_i$ of each reduction $(bexp,\gamma) \hookrightarrow^{p_i} (bexp',\gamma^i)$ to a subtree in the reduction tree, with $0 \leq i < k$, is therefore at most 1. From this and the induction hypothesis, it follows that $\mathcal{O}_{\mathcal{B}}[\![bexp]\!](\gamma)(b,\gamma') \leq 1$.

Case (branch⊥)
This transition has a derivation tree that consists of only one transition, and is probability-wise equivalent to the derivation tree of (notb), (andb1), (andb2), (orb1), (orb2) with a ($\text{branch}_\perp$) as its derivation leaf. Therefore, the proof is the exact same. □

The addition of the branch statement as a boolean expression means that a while loop can loop infinitely many times before the branch returns a `tt` path for example. This means that the sum $\mathcal{O}[\![C]\!](\gamma)(\gamma') = \sum\{p(c) \mid c \in Comp(C,\gamma,\gamma')\}$ can be infinite. We know by proposition 1 that if we take a finite partial sum from the infinite $\mathcal{O}[\![C]\!](\gamma)(\gamma')$, that it is at most 1. If we were to take the sequence of partial summations, the limit is 1, therefore proposition 1 still holds.

4.4 Overview of the Operational Semantics

Boolean Expressions

$$(v,s,G) \hookrightarrow^1 (s(v),s,G) \text{ (val)} \qquad (!b,s,G) \hookrightarrow^1 (\neg\, b,s,G) \text{ (not)}$$

$$(b \;\&\&\; b,s,G) \hookrightarrow^1 (b \wedge b,s,G) \text{ (and)} \qquad (b \;||\; b,s,G) \hookrightarrow^1 (b \vee b,s,G) \text{ (or)}$$

$$\frac{(bexp,s,G) \hookrightarrow^p (bexp',s,G')}{(!bexp,s,G) \hookrightarrow^p (!bexp',s,G')} \text{ (notb)}$$

$$\frac{(bexp_1,s,G) \hookrightarrow^p (bexp_1',s,G')}{(bexp_1 \;\&\&\; bexp_2,s,G) \hookrightarrow^p (bexp_1' \;\&\&\; bexp_2,s,G')} \text{ (andb1)}$$

$$\frac{(bexp,s,G) \hookrightarrow^p (bexp',s,G')}{(b \;\&\&\; bexp_2,s,G) \hookrightarrow^p (b \;\&\&\; bexp_2',s,G')} \text{ (andb2)}$$

$$\frac{(bexp_1,s,G) \hookrightarrow^p (bexp_1',s,G')}{(bexp_1 \;||\; bexp_2,s,G) \hookrightarrow^p (bexp_1' \;||\; bexp_2,s,G')} \text{ (orb1)} \qquad \frac{(bexp,s,G) \hookrightarrow^p (bexp',s,G')}{(b \;||\; bexp,s,G) \hookrightarrow^p (b \;||\; bexp',s,G')} \text{ (orb2)}$$

$$\frac{0 \leq i < |L| \qquad (body(L_i),G,\ \text{skip;}) \Rightarrow_b^p (G',c') \qquad b = bool(L_i)}{(\text{branch}\{L\},s,G) \hookrightarrow^{w(L_i)/w(L)\,*\,p} (b,s,G')} \text{ (branch)}$$

$$\frac{0 \leq i < |L| \qquad (body(L_i),G,\ \text{skip;}) \Rightarrow_b^{p_1} (\perp,G'',c) \qquad (c,G'',\ \text{skip;}) \Rightarrow_b^{p_2} (G',c')}{(\text{branch}\{L\},s,G) \hookrightarrow^{w(L_i)/w(L)\,*\,p_1\,*\,p_2} (\text{branch}\{L\},s,G')} \text{ (branch}_\perp)$$

Commands

$$(v = b;,\ s, G) \rightarrow^{1} (\text{skip};,\ s[v \mapsto b], G)\ \text{(assign)} \qquad \frac{(bexp, s, G) \hookrightarrow^{p} (bexp', s, G')}{(v = bexp;,\ s, G) \rightarrow^{p} (v = b;, s, G')}\ \text{(assignb)}$$

$$(\text{skip};\ C,\ s, G) \rightarrow^{1} (C,\ s, G)\ \text{(skip)} \qquad \frac{(C_1,\ s, G) \rightarrow^{p} (C_1',\ s', G')}{(C_1\ C_2,\ s, G) \rightarrow^{p} (C_1'\ C_2,\ s', G')}\ \text{(comp)}$$

$$\frac{(bexp, s, G) \hookrightarrow^{p} (\texttt{tt}, s, G')}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s, G) \rightarrow^{p} (C_1,\ s, G')}\ \text{(iftt)} \qquad \frac{(bexp, s, G) \hookrightarrow^{p} (\texttt{ff}, s, G')}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s, G) \rightarrow^{p} (C_2,\ s, G')}\ \text{(ifff)}$$

$$\frac{(bexp, s, G) \hookrightarrow^{p} (bexp', s, G')}{(\text{if}\,(bexp)\,\{C_1\}\,\text{else}\,\{C_2\},\ s, G) \rightarrow^{p} (\text{if}\,bexp'\,\{C_1\}\,\text{else}\,\{C_2\},\ s, G')}\ \text{(ifb)}$$

$$\frac{(bexp, s, G) \hookrightarrow^{p} (\texttt{tt}, s, G')}{(\text{while}\,(bexp)\,\{C\},\ s, G) \rightarrow^{p} (C\,\text{while}\,(bexp)\,\{C\},\ s, G')}\ \text{(whilett)}$$

$$\frac{(bexp, s, G) \hookrightarrow^{p} (\texttt{ff}, s, G')}{(\text{while}\,(bexp)\,\{C\},\ s, G) \rightarrow^{p} (\text{skip};,\ s, G')}\ \text{(whileff)}$$

$$\frac{(bexp, s, G) \hookrightarrow^{p} (bexp', s, G')}{(\text{while}\,(bexp)\,\{C\},\ s, G) \rightarrow^{p} (\text{while}\,(bexp')\,\{C\},\ s, G')}\ \text{(wileb)}$$

$$(\text{activate}\ \boxed{t};,\ s,\ G) \rightarrow^{1} (\text{skip};,\ s,\ \mathit{activate}(G, \boxed{t}))\ \text{(act)}$$

$$\frac{\mathcal{G} = \mathit{deactivate}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{deactivate}\ \boxed{t};\ , s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip};,\ s,\ \mathcal{G}_i)}\ \text{(deact)}$$

$$\frac{\mathcal{G} = \mathit{add}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{add}\ \boxed{t};,\ s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip};,\ s,\ \mathcal{G}_i)}\ \text{(add)} \qquad \frac{\mathcal{G} = \mathit{remove}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{remove}\ \boxed{t};,\ s,\ G) \rightarrow^{1/|\mathcal{G}|} (\text{skip};,\ s,\ \mathcal{G}_i)}\ \text{(rem)}$$

Reversible Commands

$$(\mathit{skip};, G, c) \Rightarrow_b^{1} (G, c)\ \text{(RCskip)}$$

$$\frac{\mathcal{G} = \mathit{add}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{add}\ \boxed{t};, G, c) \Rightarrow_b^{1/|\mathcal{G}|} (\mathcal{G}_i,\ \text{remove}\ \boxed{t}; + c)}\ \text{(RCadd)} \qquad \frac{\mathit{add}(G, \boxed{t}) = \{\}}{(\text{add}\ \boxed{t};, G, c) \Rightarrow_b^{1} (\bot, G, c)}\ \text{(RCadd}_\bot\text{)}$$

$$\frac{\mathcal{G} = \mathit{remove}(G, \boxed{t}) \qquad \mathcal{G} \neq \{\} \qquad 0 \leq i < |\mathcal{G}|}{(\text{remove}\ \boxed{t};, G, c) \Rightarrow_b^{1/|\mathcal{G}|} (\mathcal{G}_i,\ \text{add}\ \boxed{t}; + c)}\ \text{(RCrem)} \qquad \frac{\mathit{remove}(G, \boxed{t}) = \{\}}{(\text{remove}\ \boxed{t};, G, c) \Rightarrow_b^{1} (\bot, G, c)}\ \text{(RCrem}_\bot\text{)}$$

$$\frac{(RC_1, G, c) \Rightarrow_b^{p_1} (G'', c'') \qquad (RC_2, G'', c'') \Rightarrow_b^{p_2} (G', c')}{(RC_1 RC_2, G, c) \Rightarrow_b^{p_1 * p_2} (G', c')}\ \text{(RCcomp)}$$

$$\frac{(RC_1, G, c) \Rightarrow_b^{p_1} (\bot, G', c')}{(RC_1 RC_2, G, c) \Rightarrow_b^{p_1} (\bot, G', c')}\ \text{(RC}\bot\text{1)} \qquad \frac{(RC_1, G, c) \Rightarrow_b^{p_1} (G'', c'') \qquad (RC_2, G'', c'') \Rightarrow_b^{p_2} (\bot, G', c')}{(RC_1 RC_2, G, c) \Rightarrow_b^{p_1 * p_2} (\bot, G', c')}\ \text{(RC}\bot\text{2)}$$

5 Denotational Semantics

The denotational semantics are defined following the approach of [6], as was done for the operational semantics. The classical part of ALCH is given semantics first, then the CRN-TAM specific constructs.

5.1 Classical Constructs

For any boolean expression *bexp* its denotational meaning $\mathcal{B}[\![bexp]\!] : S \to \mathbb{B}$ is defined compositionally by induction on *bexp*.

$$\begin{aligned}
\mathcal{B}[\![b]\!] &= b \\
\mathcal{B}[\![v]\!] &= \lambda s.\ v(s) \\
\mathcal{B}[\![!bexp]\!] &= \neg\mathcal{B}[\![bexp]\!] \\
\mathcal{B}[\![bexp_1\ \&\&\ bexp_2]\!] &= \mathcal{B}[\![bexp_1]\!] \wedge \mathcal{B}[\![bexp_2]\!] \\
\mathcal{B}[\![bexp_1\ ||\ bexp_2]\!] &= \mathcal{B}[\![bexp_1]\!] \vee \mathcal{B}[\![bexp_2]\!]
\end{aligned}$$

For any command C its denotational meaning $\mathcal{C}[\![C]\!] : S \rightharpoonup S$ is defined compositionally by induction on C.

$$\begin{aligned}
\mathcal{C}[\![\text{skip;}]\!] &= id_s \\
\mathcal{C}[\![v = bexp;]\!] &= \lambda s.\ s[v \mapsto \mathcal{B}[\![bexp]\!](s)] \\
\mathcal{C}[\![C_1\ C_2]\!] &= \mathcal{C}[\![C_1]\!] \circ \mathcal{C}[\![C_2]\!] \\
\mathcal{C}[\![\text{if } (bexp)\ \{C_1\} \text{ else } \{C_2\}]\!] &= \lambda s.\ \text{if } \mathcal{B}[\![bexp]\!](s) = \texttt{tt} \text{ then } \mathcal{C}[\![C_1]\!](s) \text{ else } \mathcal{C}[\![C_2]\!](s) \\
\mathcal{C}[\![\text{while } (bexp)\ \{C\}]\!] &= \text{lfp}\,[\lambda f.\ \lambda s.\ \text{if } \mathcal{B}[\![bexp]\!](s) = \texttt{tt} \text{ then } (f \circ \mathcal{C}[\![C]\!])(s) \text{ else } s]
\end{aligned}$$

The set of partial functions $[S \rightharpoonup S]$ is a CPO (complete partial order), with, given partial functions $g : S \rightharpoonup S$ and $h : S \rightharpoonup S$, an ordering $h \sqsubseteq g$ such that $dom(h) \subseteq dom(g)$ and $\forall s \in dom(h).\ h(s) = g(s)$ [6]. Furthermore, $[S \rightharpoonup S]$, given that S is not empty, we know that is a $\text{CPO}_\perp$. Its least element is the function with the empty domain. The function between the square brackets uses the composition operator, which is essentially a function $\circ : ((S \rightharpoonup S) \times (S \rightharpoonup S)) \to (S \rightharpoonup S)$. This composition is separately continuous, therefore it is continuous itself ([16], Theorem 3.2.6). Any continuous function on a $\text{CPO}_\perp$ has a least fixpoint by Kleene's fixpoint theory ([5], Theorem 5.6).

5.2 CRN-TAM Constructs

With the introduction of a new graph-like object $G \in \mathbb{G}$ as an additional state, denotational meanings are now of the form $\mathcal{C}[\![C]\!] : \Gamma \rightharpoonup \Gamma$, where Γ is shorthand for $(\mathbb{G} \times S)$. In this section the graph operators defined in Section 4 will be used as well. Due to the non-deterministic nature of CRN-TAM commands, for any command C we use the sub-distribution monad $\mathcal{D}$, where $\mathcal{D}(\Gamma)$ can be seen as the set of functions $\omega : \Gamma \to [0,1]$ such that $\sum_{\gamma \in \Gamma} \omega(\gamma) \leq 1$.

Any partial function $f : \Gamma \rightharpoonup \Gamma$ can be embedded in $\Gamma \to \mathcal{D}(\Gamma)$ using the unit operator $\eta^{\mathcal{D}}_{\Gamma}$ as follows: $\eta^{\mathcal{D}}_{\Gamma}(f(\gamma)) : \mathcal{D}(\Gamma)$. A function $\bar{g}(g) : \Gamma \to \mathcal{D}(\Gamma)$ is defined that takes a function g from graphs to sets of graphs and is defined as:

$$\bar{g}(g)(s,G)(s',G') = \begin{cases} 1/|g(G)| & \text{if } G' \in g(G) \\ 0 & \text{otherwise,} \end{cases}$$

where $|g(G)|$ is the size of the set returned by $g(G)$.

$$\begin{aligned}
\mathcal{C}[\![\text{activate } \boxed{t};]\!] &= \lambda s.\ \lambda G.\ \eta^{\mathcal{D}}_{\Gamma}(s, activate(G, \boxed{t})) \\
\mathcal{C}[\![\text{deactivate } \boxed{t};]\!] &= \bar{g}(g) \text{ where } g = \lambda G.\ deactivate(G, \boxed{t}) \\
\mathcal{C}[\![\text{add } \boxed{t};]\!] &= \bar{g}(g) \text{ where } g = \lambda G.\ add(G, \boxed{t}) \\
\mathcal{C}[\![\text{remove } \boxed{t};]\!] &= \bar{g}(g) \text{ where } g = \lambda G.\ remove(G, \boxed{t})
\end{aligned}$$

The previously defined denotational semantics need to be redefined so that they too return a probability distribution. Because the co-domain of the semantics are now a different type than its domain, standard composition of commands is not possible. Given the sub-distribution monad $\mathcal{D}$ we can use Kleisli category $\mathcal{K}\ell(\mathcal{D})$. The Kleisli composition operator enables us to compose two functions $f : \Gamma \to \mathcal{D}(\Gamma)$ and $g : \Gamma \to \mathcal{D}(\Gamma)$ like so: $g \circledcirc f$. To denote the Kleisli composition of $\mathcal{K}\ell(\mathcal{D})$ the symbol $\circledcirc^{\mathcal{D}}$ is used.

$$\begin{aligned}
\mathcal{C}[\![\text{skip};]\!] &= \eta^{\mathcal{D}}_{\Gamma} \\
\mathcal{C}[\![v = bexp;]\!] &= \lambda s.\ \lambda G.\ \eta^{\mathcal{D}}_{\Gamma}(s[v \mapsto \mathcal{B}[\![bexp]\!](s)], G) \\
\mathcal{C}[\![C_1\ C_2]\!] &= \mathcal{C}[\![C_2]\!] \circledcirc^{\mathcal{D}} \mathcal{C}[\![C_1]\!] \\
\mathcal{C}[\![\text{if } (bexp)\ \{C_1\} \text{ else } \{C_2\}]\!] &= \lambda\gamma.\ \text{if } \mathcal{B}[\![bexp]\!](\gamma) = \texttt{tt} \text{ then } \mathcal{C}[\![C_1]\!](\gamma) \text{ else } \mathcal{C}[\![C_2]\!](\gamma) \\
\mathcal{C}[\![\text{while } (bexp)\ \{C\}]\!] &= lpf[\lambda f.\ \lambda\gamma.\ \text{if } \mathcal{B}[\![bexp]\!](\gamma) = \texttt{tt} \text{ then } (f \circledcirc^{\mathcal{D}} \mathcal{C}[\![C]\!])(\gamma) \text{ else } \eta^{\mathcal{D}}_{\Gamma}(\gamma)]
\end{aligned}$$

The Kleisli category on the sub-distribution monad $\mathcal{D}$ is a $\text{CPO}_\perp$ ([10], Lemma 2.5), with an ordering $h \sqsubseteq g$ on morphisms $h : X \to \mathcal{D}(Y)$ and $g : X \to \mathcal{D}(Y)$ such that $\forall x \in X.\ \forall y \in Y.\ h(x)(y) \le g(x)(y)$, where $X, Y \in \textbf{Sets}$. The Kleisli composition can be seen as a function $\circledcirc^{\mathcal{D}} : ((X \to \mathcal{D}(Y)) \to (X \to \mathcal{D}(Y))) \to (X \to \mathcal{D}(Y))$, this function is separately continuous. Therefore the Kleisli composition, and by extension the funciton between the square brackets, is continuous ([16], Theorem 3.2.6). Given that the function between the square brackets is a continuous function on a $\text{CPO}_\perp$, according to Kleene's fixpoint theory it has a least fixpoint ([5], Theorem 5.6).

5.3 Some More Monads

The operational semantics of the reversible commands return whether the execution of its path was successful or not. This approach will be replicated in the denotational semantics using the set $\mathbf{2} = \{\top, \perp\}$. The semantics for the reversible commands will be of the form $f : X \to \mathcal{D}(X \times \mathbf{2})$. When composing reversible commands a $\perp$ element should always be returned if a single reversible command in the composition fails. Simply using the Kleisli composition $\circledcirc^{\mathcal{D}}$ of $\mathcal{K}\ell(\mathcal{D})$ does not achieve this, it always preserves the last $b \in \mathbf{2}$ in the composition. We define a new monad $\mathcal{DE}$, along with the operation φ and the monad $\mathcal{E}$, that will allow for a composition $\circledcirc^{\mathcal{DE}}$ which preserves $\perp$.

Definition 19. The monad $\mathcal{E}$ on **Sets** is, given a set X, defined as $\mathcal{E}(X) = X \times \mathbf{2}$. It has the unit transformation $\eta^{\mathcal{E}}_X(x) = (x, \top)$ and the multiplication transformations $\mu^{\mathcal{E}}_X((x, b), b') = (x, b \wedge b')$.

This monad *extends* all elements in a set with a $\top$ or $\perp$. Its multiplication transformation will only return $(x, \top)$ when both $b, b' \in \mathbf{2}$ are $\top$ also, in all other cases it returns $(x, \perp)$.

Definition 20. The operator $\varphi_X : \mathcal{D}(X) \times 2 \to \mathcal{D}(X \times 2)$ is defined as $\varphi_X(d, b)(x, b') = \begin{cases} d(x), & \text{if } b = b' \\ 0, & \text{otherwise.} \end{cases}$

The monad $\mathcal{E}$ and the operator φ allow us to define the natural transformations of the distribution extension monad $\mathcal{DE}$.

Definition 21. The monad $\mathcal{DE}$ on **Sets** is, given a set X, defined as $\mathcal{DE}(X) = \mathcal{D}(X \times 2)$. Its natural transformations are defined as (see also Figure 10 and Figure 11):
- unit transformation: $\eta^{\mathcal{DE}}_X(x) = (\varphi_x \circ \eta^{\mathcal{E}}_{\mathcal{D}(X)} \circ \eta^{\mathcal{D}}_X)(x)$
- multiplication transformation: $\mu^{\mathcal{DE}}_X = \mathcal{D}(\mu^{\mathcal{E}}_X) \circ \mu^{\mathcal{D}}_{X \times 2 \times 2} \circ \mathcal{D}(\varphi_{(X \times 2)})$

$$\begin{array}{ccccc}
X & \xrightarrow{\eta^{\mathcal{D}}_X} & \mathcal{D}(X) & \xrightarrow{\eta^{\mathcal{E}}_{\mathcal{D}(X)}} & \mathcal{D}(X) \times \mathbf{2} \\
 & \searrow^{\eta^{\mathcal{DE}}_X} & & & \downarrow \varphi_X \\
 & & & & \mathcal{D}(X \times \mathbf{2})
\end{array}$$

Figure 10: Commutativity diagram of the unit transformation of the monad $\mathcal{DE}$

$$\begin{array}{ccc} \mathcal{D}(\mathcal{D}(X \times \mathbf{2}) \times \mathbf{2}) & \xrightarrow{\mathcal{D}(\varphi(X\times\mathbf{2}))} & \mathcal{D}(\mathcal{D}(X \times \mathbf{2} \times \mathbf{2})) \\ & & \downarrow \mu^{\mathcal{D}}_{X\times\mathbf{2}\times\mathbf{2}} \\ & \searrow^{\mu^{\mathcal{DE}}_X} & \mathcal{D}(X \times \mathbf{2} \times \mathbf{2}) \\ & & \downarrow \mathcal{D}(\mu^{\mathcal{E}}_X) \\ & & \mathcal{D}(X \times \mathbf{2}) \end{array}$$

Figure 11: Commutativity diagram of the multiplication transformation of the monad $\mathcal{DE}$

The distribution extension monad $\mathcal{DE}$ allows for the use of the Kleisli category $\mathcal{K}\ell(\mathcal{DE})$ and the Kleisli composition $\circledcirc^{\mathcal{DE}}$. Now, two functions $f : X \to \mathcal{D}(X \times \mathbf{2})$ and $g : X \to \mathcal{D}(X \times \mathbf{2})$ can be composed as $g \circledcirc^{\mathcal{DE}} f = \mu^{\mathcal{DE}}_x \circ \mathcal{DE}(g) \circ f$. Figure 12 depicts how $\mu^{\mathcal{DE}}_x$ returns a single extended distribution from nested extended distributions, such as the ones returned by $\mathcal{DE}(g)$. The three steps in this example correspond to the arrows in Figure 11.

$$\begin{bmatrix} 0.5 & \begin{bmatrix} 0.5 & x1 & \top \\ 0.5 & x2 & \bot \end{bmatrix} & \top \\ 0.5 & \begin{bmatrix} 0.8 & x3 & \top \\ 0.2 & x4 & \bot \end{bmatrix} & \bot \end{bmatrix} : \mathcal{D}(\mathcal{D}(X \times \mathbf{2}) \times \mathbf{2})$$

$$\begin{bmatrix} 0.5 & \begin{bmatrix} 0.5 & x1 & \top & \top \\ 0.5 & x2 & \bot & \top \end{bmatrix} \\ 0.5 & \begin{bmatrix} 0.8 & x3 & \top & \bot \\ 0.2 & x4 & \bot & \bot \end{bmatrix} \end{bmatrix} : \mathcal{D}(\mathcal{D}(X \times \mathbf{2} \times \mathbf{2}))$$

$$\begin{bmatrix} 0.25 & x1 & \top & \top \\ 0.25 & x2 & \bot & \top \\ 0.4 & x3 & \top & \bot \\ 0.1 & x4 & \bot & \bot \end{bmatrix} : \mathcal{D}(X \times \mathbf{2} \times \mathbf{2})$$

$$\begin{bmatrix} 0.25 & x1 & \top \\ 0.25 & x2 & \bot \\ 0.4 & x3 & \bot \\ 0.1 & x4 & \bot \end{bmatrix} : \mathcal{D}(X \times \mathbf{2})$$

Figure 12: Example of how a nested distribution $\mathcal{D}(\mathcal{D}(X \times \mathbf{2}) \times \mathbf{2})$ is mapped to a distribution $\mathcal{D}(X \times \mathbf{2})$ in steps

The evaluation of boolean expressions is done by functions of the form $f : X \to \mathcal{D}(\mathbb{B} \times X)$. In the operational semantics, the evaluation of and-expressions and or-expressions is little more than simply composing the left and right side of the expression. Of course, the evaluation ends only when the resulting boolean values have been combined properly. In order to compose the sub-expressions of and-expressions and or-expressions, we require similar monads to $\mathcal{DE}$.

Both monads will use the following operator.

Definition 22. The operator $\beta_X : \mathbb{B} \times \mathcal{D}(X) \to \mathcal{D}(\mathbb{B} \times X)$ is defined as $\beta_X(b, d)(b', x) = \begin{cases} d(x), & \text{if } b = b' \\ 0, & \text{otherwise} \end{cases}$

First, lets define the monad to evaluate and-expressions. This requires a sort of extender monad which allows us to extend an element with a boolean value.

Definition 23. The monad A on **Sets** is, given a set X, defined as $A(X) = \mathbb{B} \times X$. It has a unit transformation $\eta^A_X(x) = (\mathtt{tt}, x)$ and a multiplication transformation $\mu^A_X(b, (b', x)) = (b \wedge b', x)$.

Now the monad $\mathcal{D}A$ for and-expressions can be defined.

Definition 24. The monad $\mathcal{D}A$ on **Sets** is, given a set X, defined as $\mathcal{D}A(X) = \mathcal{D}(\mathbb{B} \times X)$. Its natural transformations are defined as:
- unit transformation: $\eta^{\mathcal{D}A}_X = (\beta_X \circ \eta^A_{\mathcal{D}(X)} \circ \eta^{\mathcal{D}}_X)(x)$
- multiplication transformation: $\mu^{\mathcal{D}A}_X = \mathcal{D}(\mu^A_X) \circ \mu^{\mathcal{D}}_{\mathbb{B}\times\mathbb{B}\times X} \circ \mathcal{D}(\beta(\mathbb{B} \times X))$

Monad $\mathcal{D}O$ is similar to monad $\mathcal{D}A$.

Definition 25. The monad O on **Sets** is, given a set X, defined as $O(X) = \mathbb{B} \times X$. It has a unit transformation $\eta^O_X(x) = (\mathtt{ff}, x)$ and a multiplication transformation $\mu^O_X(b, (b', x)) = (b \vee b', x)$.

Definition 26. The monad $\mathcal{D}O$ on **Sets** is, given a set X, defined as $\mathcal{D}O(X) = \mathcal{D}(\mathbb{B} \times X)$. Its natural transformations are defined as:
- unit transformation: $\eta^{\mathcal{D}O}_X = (\beta_X \circ \eta^O_{\mathcal{D}(\mathcal{X})} \circ \eta^{\mathcal{D}}_X)(x)$
- multiplication transformation: $\mu^{\mathcal{D}O}_X = \mathcal{D}(\mu^O_X) \circ \mu^{\mathcal{D}}_{\mathbb{B}\times\mathbb{B}\times X} \circ \mathcal{D}(\beta(\mathbb{B} \times X))$

5.4 The Branch Construct

Introducing the branch expression to the semantics of the boolean expressions causes it to also return a sub-probability distribution. The denotational semantics of boolean expressions are now $\mathcal{B}[\![bexp]\!] : \Gamma \to \mathcal{D}(\mathbb{B} \times \Gamma)$ with distribution monad $\mathcal{D}$ such that $\mathcal{D}(\mathbb{B} \times \Gamma)$ is the set of functions $\omega_B : \mathbb{B} \times \Gamma \to [0, 1]$ s.t. $\sum_{(b,\gamma)\in\mathbb{B}\times\Gamma} \omega_B(b, \gamma) \leq 1$.

$$\begin{aligned}
\mathcal{B}[\![b]\!] &= \lambda\gamma.\ \eta^{\mathcal{D}}_{\mathbb{B}\times\Gamma}(b, \gamma) \\
\mathcal{B}[\![v]\!] &= \lambda s.\ \lambda G.\ \eta^{\mathcal{D}}_{\mathbb{B}\times\Gamma}(s(v), (s, G)) \\
\mathcal{B}[\![!bexp]\!] &= \lambda\gamma.\ \lambda b.\ \lambda\gamma'.\ \mathcal{B}[\![bexp]\!](\gamma)(\neg b, \gamma') \\
\mathcal{B}[\![bexp_1 \&\& bexp_2]\!] &= \mathcal{B}[\![bexp_2]\!] \circledcirc^{\mathcal{D}A} \mathcal{B}[\![bexp_1]\!] \\
\mathcal{B}[\![bexp_1 || bexp_2]\!] &= \mathcal{B}[\![bexp_2]\!] \circledcirc^{\mathcal{D}O} \mathcal{B}[\![bexp_1]\!]
\end{aligned}$$

Reversible Commands
Before the semantics for the branch expression can be defined, the semantics for the reversible commands $\mathcal{RC}[\![RC]\!](\mathbb{G} \times RC^*) \to \mathcal{D}(\mathbb{G} \times RC^* \times \mathbf{2})$ should be given. Here, RC^* is the set of lists of reversible commands. To calculate the probability corresponding to the execution of a path we define a function $\tilde{g}(g)(l) : (\mathbb{G} \times RC^*) \to (\mathbb{G} \times RC^* \times \mathbf{2})$, that takes a function $g : \mathbb{G} \to \mathcal{P}(\mathbb{G})$ and a function $l : RC^* \to RC^*$, as:

$$\tilde{g}(g)(l)(G, c)(G', c', b) = \begin{cases} 1/|g(G)|, & \text{if } G' \in g(G), l(c) = c', b = \top \\ 1, & \text{if } g(G) = \{\}, c = c', G = G', b = \bot \\ 0, & \text{otherwise.} \end{cases}$$

In addition, we define the function $compose : \mathbb{G} \times RC^* \to \mathcal{D}(\mathbb{G} \times RC^* \times \mathbf{2})$ as

$$compose(G, c)(G', c', b) = \begin{cases} (\mathcal{RC}[\![RC_2]\!] \circledcirc^{\mathcal{DE}} \mathcal{RC}[\![RC_1]\!])(G, c)(G', c', b), & \text{if } b = \top \\ \mathcal{RC}[\![RC_1]\!](G, c)(G', c', b)\ + \\ (\mathcal{RC}[\![RC_2]\!] \circledcirc^{\mathcal{DE}} \mathcal{RC}[\![RC_1]\!])(G, c)(G', c', b), & \text{if } b = \bot. \end{cases}$$

Using these operators, we can define the denotational semantics of the reversible commands.

$$\begin{aligned}
\mathcal{RC}[\![\text{skip};]\!] &= \eta^{\mathcal{DE}}_{\mathbb{G}\times RC^*}\\
\mathcal{RC}[\![\text{add } \boxed{t};]\!] &= \tilde{g}(g)(l) \text{ where } g = \lambda G.\ add(G, \boxed{t}),\ l = \lambda c.\ remove\ \boxed{t}; +c\\
\mathcal{RC}[\![\text{remove } \boxed{t};]\!] &= \tilde{g} \text{ where } g = \lambda G.\ remove(G, \boxed{t}),\ l = \lambda c.\ add\ \boxed{t}; +c\\
\mathcal{RC}[\![RC_1\ RC_2]\!] &= \lambda G.\ \lambda c.\ \lambda G'.\ \lambda c'.\ compose(G, c)(G', c')
\end{aligned}$$

Branch Expression
The semantics of reversible commands $\mathcal{RC}[\![RC]\!] : \mathbb{G} \times RC^* \to \mathcal{D}(\mathcal{D} \times RC^* \times \mathbf{2})$ are used to execute the individual branch paths. Every branch path has a corresponding boolean value, to account for this a function $B : \mathbb{B} \times \mathcal{D}(\mathbb{G} \to RC^* \to \mathbf{2}) \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2})$ is defined which adds this boolean value to any distribution as follows:

$$B(b, d)(b', x) = \begin{cases} d(x), & \text{if } b = b' \\ 0, & \text{otherwise.} \end{cases}$$

Here, x is a value $(G, c, \bot)$ or $(G, c, \top)$.

To combine the distributions from each path, a union operator is used, which takes distributions and weights as arguments, and multiplies each weight with each distribution $\bigsqcup : ([0,1] \times \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2}))^{\mathbb{N}} \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2})$. Note that the union operator returns a sub-distribution if and only if the sum of given weights is smaller than or equal to 1 ([17], Lemma 2).

We now define a function $r(L) : \mathbb{G} \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times 2)$ as follows:

$$r(L)(G) = \bigsqcup_{i=0}^{|L|} (w(L_i))(B(bool(L_i))(\mathcal{RC}[\![body(L_i)]\!](G,\ \text{skip};))).$$

This function finds the distribution over the single execution of all branch paths. In this distribution, every $\bot$ outcome needs to be reversed, if reversing is successful the branch is evaluated once more. We define a function $cont(h) : (\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2}) \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2})$ that continues a successfully reversed path as follows

$$cont(h)(b, G', c, \top) = h(G')$$
$$cont(h)(b, G', c, \bot) = \eta^{\mathcal{D}}(b, G', c, \bot).$$

Note that every $\bot$ outcome is not explored further, since its reversing has failed. Using $cont(h)$, we can now define a function $retry(h) : (\mathbb{B} \times \mathbb{G} \times RC^* \times 2) \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2})$ as follows

$$retry(h)(b, G', c, \top) = \eta^{\mathcal{D}}(b, G', c, \top)$$

$$retry(h)(b, G', c, \bot) = cont(h) \circledcirc^{\mathcal{D}} B(b)(\mathcal{RC}[\![c]\!](G',\ \text{skip};)).$$

The approach for defining these operations is based on the operational semantics, if a path fails ($\bot$) is it reversed using $\mathcal{RC}[\![c]\!](G', \epsilon)$. The branch is only re-evaluated if the reversion is successful ($\top$)

Given a list L a least fixpoint function can be defined:

$$lfp[\lambda h : \mathbb{G} \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2}).\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L))(G)].$$

The function between square brackets has a least fixpoint since it is a continuous function on a $\text{CPO}_\bot$ ([5], Theorem 5.6). This we know because the function is nothing but a Kleisli composition on another Kleisli composition, which both are separately continuous, therefore the entire function is continuous ([16], Theorem 3.2.6).

The outcome of the least fixpoint function is $\mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times \mathbf{2})$, while the only relevant parts of this outcome, for the branch semantics as a whole, are the boolean value and the graph outcome. We define a function $convert : \Gamma \to \mathcal{D}(\mathbb{B} \times \mathbb{G} \times RC^* \times 2) \to \mathcal{D}(\mathbb{B} \times \Gamma)$ as

$$convert(s, G)(d)(b, s', G') = \hat{id}_S(s)(s') \cdot \sum_c d(b, G', c, \top).$$

This allows us to define the branch semantics.

$$\mathcal{B}[\![\text{branch}(L)]\!] = \lambda s.\ \lambda G.\ convert(s, G)(lfp[\lambda h.\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L))(G)])$$

Commands

It is necessary to redefine the semantics of the commands that include a boolean expression. Given the functions $f : \Gamma \to \mathcal{D}(\Gamma)$ and $g : \Gamma \to \mathcal{D}(\Gamma)$ we can define the co-pairing $[f, g] : \mathbb{B} \times \Gamma \to \mathcal{D}(\Gamma)$ as:

$$[f, g](\mathtt{tt}, \gamma) = f(\gamma)$$

$$[f, g](\mathtt{ff}, \gamma) = g(\gamma).$$

Here the domain $\mathbb{B} \times \Gamma$ can be seen as the co-product $\Gamma + \Gamma$, which exists in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$. The co-pairing $[f, g]$ allows for the use of the Kleisli composition, given any function $h : \Gamma \to \mathcal{D}(\mathbb{B} \times \Gamma)$, like so:

$$([f, g] \circledcirc^{\mathcal{D}} h)(\gamma)(\gamma') = \sum_{\gamma'' \in \Gamma} h(\gamma)(\mathtt{tt}, \gamma'') \cdot f(\gamma'')(\gamma') + h(\gamma)(\mathtt{ff}, \gamma'') \cdot g(\gamma'')(\gamma').$$

In addition, a function $n(b) : \Gamma \to \mathcal{D}(\Gamma)$ is defined as $n(b)(s, G) = \eta^{\mathcal{D}}_{\Gamma}(s[v \mapsto b], G)$. These functions allows us to re-define the semantics of the following commands.

$$\begin{aligned}
\mathcal{C}[\![v = bexp;]\!] &= [n(\mathtt{tt}), n(\mathtt{ff})] \circledcirc^{\mathcal{D}} \mathcal{B}[\![bexp]\!] \\
\mathcal{C}[\![\text{if } (bexp) \text{ then } \{C_1\} \text{ else } \{C_2\}]\!] &= [\mathcal{C}[\![C_1]\!], \mathcal{C}[\![C_2]\!]] \circledcirc^{\mathcal{D}} \mathcal{B}[\![bexp]\!] \\
\mathcal{C}[\![\text{while } (bexp)\{C\}[\![&= lfp[\lambda f.\ [f \circledcirc^{\mathcal{D}} \mathcal{C}[\![C]\!], \eta^{\mathcal{D}}_{\Gamma}] \circledcirc^{\mathcal{D}} \mathcal{B}[\![bexp]\!]]
\end{aligned}$$

Given that the co-pairing is argument-wise continuous and the Kleisli composition is separately continuous, the function between square brackets is itself continuous ([16], Theorem 3.2.6). Since it is a continuous function on a $\text{CPO}_\perp$, it has a least fixpoint ([5], Theorem 5.6).

5.5 Overview of the Denotational Semantics

Boolean Expressions

$$\begin{aligned}
\mathcal{B}[\![b]\!] &= \lambda\gamma.\ \eta^{\mathcal{D}}_{\mathbb{B}\times\Gamma}(\gamma, b) \\
\mathcal{B}[\![v]\!] &= \lambda s.\ \lambda G.\ \eta^{\mathcal{D}}_{\mathbb{B}\times\Gamma}(s(v), (s, G)) \\
\mathcal{B}[\![!bexp]\!] &= \lambda\gamma.\ \lambda b.\ \mathcal{B}[\![bexp]\!](\gamma)(\neg b) \\
\mathcal{B}[\![bexp_1 \& \& bexp_2]\!] &= \mathcal{B}[\![bexp_2]\!] \circledcirc^{\mathcal{D}A} \mathcal{B}[\![bexp_1]\!] \\
\mathcal{B}[\![bexp_1 | | bexp_2]\!] &= \mathcal{B}[\![bexp_2]\!] \circledcirc^{\mathcal{D}O} \mathcal{B}[\![bexp_1]\!] \\
\mathcal{B}[\![\text{branch}(L)]\!] &= \lambda s.\ \lambda G.\ convert(s, G)(lfp[\lambda h.\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L))(G)])
\end{aligned}$$

Commands

$$
\begin{aligned}
\mathcal{C}[\![\text{skip;}]\!] &= \eta^{\mathcal{D}}_{\Gamma}\\
\mathcal{C}[\![v = bexp;]\!] &= [n(\texttt{tt}), n(\texttt{ff})] \circledcirc^{\mathcal{B}} \mathcal{B}[\![bexp]\!]\\
\mathcal{C}[\![C_1\ C_2]\!] &= \mathcal{C}[\![C_2]\!] \circledcirc^{\mathcal{D}} \mathcal{C}[\![C_1]\!]\\
\mathcal{C}[\![\text{if } (bexp) \text{ then } \{C_1\} \text{ else } \{C_2\}]\!] &= [\mathcal{C}[\![C_1]\!], \mathcal{C}[\![C_2]\!]] \circledcirc^{\mathcal{D}} \mathcal{B}[\![bexp]\!]\\
\mathcal{C}[\![\text{while } (bexp)\{C\}[\![&= lfp[\lambda f.\ [f \circledcirc^{\mathcal{D}} \mathcal{C}[\![C]\!], \eta^{\mathcal{D}}_{\Gamma}] \circledcirc^{\mathcal{D}} \mathcal{B}[\![bexp]\!]]\\
\mathcal{C}[\![\text{activate } \boxed{t};]\!] &= \hat{f} \text{ where } f = \lambda s.\ \lambda G.\ (s,\ activate(G, \boxed{t}))\\
\mathcal{C}[\![\text{deactivate } \boxed{t};]\!] &= \bar{g} \text{ where } g = \lambda G.\ deactivate(G, \boxed{t})\\
\mathcal{C}[\![\text{add } \boxed{t};]\!] &= \bar{g} \text{ where } g = \lambda G.\ add(G, \boxed{t})\\
\mathcal{C}[\![\text{remove } \boxed{t};]\!] &= \bar{g} \text{ where } g = \lambda G.\ remove(G, \boxed{t})
\end{aligned}
$$

Reversible Commands

$$
\begin{aligned}
\mathcal{RC}[\![\text{skip;}]\!] &= \eta^{\mathcal{DE}}_{\mathbb{G}\times RC^*}\\
\mathcal{RC}[\![\text{add } \boxed{t};]\!] &= \tilde{g}(g)(l) \text{ where } g = \lambda G.\ add(G, \boxed{t}),\ l = \lambda c.\ remove\ \boxed{t}; +c\\
\mathcal{RC}[\![\text{remove } \boxed{t};]\!] &= \tilde{g}(g)(l) \text{ where } g = \lambda G.\ remove(G, \boxed{t}),\ l = \lambda c.\ add\ \boxed{t}; +c\\
\mathcal{RC}[\![RC_1\ RC_2]\!] &= \lambda G.\ \lambda c.\ \lambda G'.\ \lambda c'.\ compose(G, c)(G', c')
\end{aligned}
$$

Helper Functions

$$
\begin{aligned}
\bar{g}(g)(s, G)(s', G') &= \begin{cases} 1/|g(G)| & \text{if } G' \in g(G)\\ 0 & \text{otherwise,} \end{cases}\\
\tilde{g}(g)(l)(G, c)(G', c', b) &= \begin{cases} 1/|g(G)|, & \text{if } G' \in g(G), l(c) = c', b = \top\\ 1, & \text{if } g(G) = \{\}, c = c', G = G', b = \bot\\ 0, & \text{otherwise.} \end{cases}\\
compose(G, c)(G', c', b) &= \begin{cases} (\mathcal{RC}[\![RC_2]\!] \circledcirc^{\mathcal{DE}} \mathcal{RC}[\![RC_1]\!])(G, c)(G', c', b), & \text{if } b = \top\\ \mathcal{RC}[\![RC_1]\!](G, c)(G', c', b)\ + & \\ (\mathcal{RC}[\![RC_2]\!] \circledcirc^{\mathcal{DE}} \mathcal{RC}[\![RC_1]\!])(G, c)(G', c', b), & \text{if } b = \bot. \end{cases}\\
n(b)(s, G) &= \eta^{\mathcal{D}}_{\Gamma}(s[v \mapsto b], G)\\
r(L)(G) &= \bigsqcup_{i=0}^{|L|} (w(L_i))(B(bool(L_i))(\mathcal{RC}[\![body(L_i)]\!](G,\ \text{skip;})))\\
cont(h)(b, G', c, \top) &= h(G')\\
cont(h)(b, G', c, \bot) &= \eta^{\mathcal{D}}(b, G', c, \bot)\\
retry(h)(b, G', c, \top) &= \eta^{\mathcal{D}}(b, G', c, \top)\\
retry(h)(b, G', c, \bot) &= cont(h) \circledcirc^{\mathcal{D}} B(b)(\mathcal{RC}[\![c]\!](G',\ \text{skip;}))\\
convert(s, G)(d)(b, s', G') &= \hat{id}_S(s)(s') \cdot \sum_{c} d(b, G', c, \top)
\end{aligned}
$$

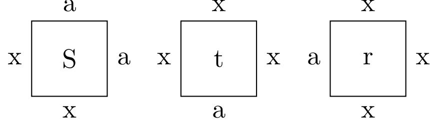
5.6 Examples

In this section some examples will be given to demonstrate the denotational semantics for the branch statement. The first example includes a branch expression of which all paths can be executed correctly in the

first attempt. The second example includes a branch expression which has no paths that can execute, but that can always be reversed.

Example 1

Temperature: $\tau = 2$
Bondlabels: $a = (a, 2), x = (x, 0)$
Tiles:



ALCH code:

```
1    activate S;
2    v = branch {
3        true (2) { add t; }
4        false (1) { add r; }
5    }
```

The initial state of any ALCH program has no assemblies present yet. The initial graph-like object G^0 contains no nodes, seed tiles or edges. The first construct to be evaluated is the activation, which is deterministic.

`1.` $\mathcal{C}[\![\text{activate } \boxed{S};]\!](s^0, G^0) \;=\; \begin{bmatrix} 1.0 & s^0 & \boxed{S} \end{bmatrix}$

The second construct to be evaluated is an assignment:

`2.` $\mathcal{C}[\![v = bexp\boxed{S};]\!](s^0, \boxed{S})$

which includes a boolean branch expression. The fist step of evaluating a branch expression is to evaluate its reversible paths:

`tt path (3).` $\mathcal{RC}[\![\text{ add } \boxed{t};]\!](\boxed{S}, \epsilon) \;=\; \begin{bmatrix} 1.0 & \begin{smallmatrix}\boxed{t}\\\boxed{S}\end{smallmatrix} & \text{remove } \boxed{t}; & \top \end{bmatrix}$

`ff path (4).` $\mathcal{RC}[\![\text{ add } \boxed{r};]\!](\boxed{S}, \epsilon) \;=\; \begin{bmatrix} 1.0 & \boxed{S}\boxed{r} & \text{remove } \boxed{r}; & \top \end{bmatrix}$

and joining the resulting distributions using the corresponding path weights.

`join.` $r(\{2,\ \texttt{tt},\ \text{add } \boxed{t};\}, \{1,\ \texttt{ff},\ \text{remove } \boxed{t};\})(\boxed{S}) \;=\; \begin{bmatrix} 0.67 & \texttt{tt} & \begin{smallmatrix}\boxed{t}\\\boxed{S}\end{smallmatrix} & \text{remove } \boxed{t}; & \top \\ 0.33 & \texttt{ff} & \boxed{S}\boxed{r} & \text{remove } \boxed{r}; & \top \end{bmatrix}$

Recall that the denotational semantics for the branch statement includes a fixpoint calculation
$\lambda h.\, \lambda G.\, (retry(h) \circledcirc^{\mathcal{D}} r(L))(G)$.

`fixpoint.` $(\lambda h.\, \lambda G.\, (retry(h) \circledcirc^{\mathcal{D}} r(L)))(\boxed{S}) \;=\; \lambda h.\, (retry(h) \circledcirc^{\mathcal{D}} \begin{bmatrix} 0.67 & \texttt{tt} & \begin{smallmatrix}\boxed{t}\\\boxed{S}\end{smallmatrix} & \text{remove } \boxed{t}; & \top \\ 0.33 & \texttt{ff} & \boxed{S}\boxed{r} & \text{remove } \boxed{r}; & \top \end{bmatrix}) \;=$

$$\lambda h.\, \eta^{\mathcal{D}}_{\mathrm{G}} \circledcirc^{\mathcal{D}} \begin{bmatrix} 0.67 & \texttt{tt} & \begin{smallmatrix}\boxed{t}\\\boxed{S}\end{smallmatrix} & \text{remove } \boxed{t}; & \top \\ 0.33 & \texttt{ff} & \boxed{S}\boxed{r} & \text{remove } \boxed{r}; & \top \end{bmatrix} \;=\; \lambda h.\, \begin{bmatrix} 0.67 & \texttt{tt} & \begin{smallmatrix}\boxed{t}\\\boxed{S}\end{smallmatrix} & \text{remove } \boxed{t}; & \top \\ 0.33 & \texttt{ff} & \boxed{S}\boxed{r} & \text{remove } \boxed{r}; & \top \end{bmatrix}$$

Both paths, given initial state $\boxed{S}$, execute successfully in the first "loop" of the branch expression. This means that the function h is never called. After converting the resulting distribution, we have evaluated the branch expression.

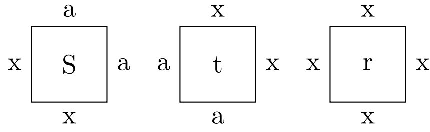
$$\texttt{bool.}\quad \mathcal{B}[\![\text{branch\{ true(2) \{add } \boxed{t}\text{;\} false(1) \{add } \boxed{r}\text{;\}}]\!](s, \boxed{S}) \;=\; \begin{bmatrix} & & & \boxed{t} \\ 0.67 & \texttt{tt} & s^0 & \boxed{S} \\ 0.33 & \texttt{ff} & s^0 & \boxed{S}\boxed{r} \end{bmatrix}$$

From this follows the evaluation of the assignment:

$$\texttt{2.}\quad \mathcal{C}[\![v = \text{branch(L)}]\!](s^0, \boxed{S}) = \begin{bmatrix} & & \boxed{t} \\ 0.67 & s^1 = s^0[v \mapsto \texttt{tt}] & \boxed{S} \\ 0.33 & s^2 = s^0[v \mapsto \texttt{ff}] & \boxed{S}\boxed{r} \end{bmatrix}$$

Example 2

Temperature: $\tau = 2$
Bondlabels: $a = (a, 2),\ x = (x, 0)$
Tiles:



ALCH code:

```
1    activate S;
2    v = branch {
3        true(1) { add t; add r; }
4        false(1) { add r; }
5    }
```

The begin of this example is the same as the previous, therefore we skip to the reversible commands immediately. The first path consist of a composition of two reversible commands.

$$\mathcal{RC}[\![\text{add } \boxed{t};]\!](\boxed{S}, \epsilon) \;=\; \begin{bmatrix} & \boxed{t} & & \\ 0.5 & \boxed{S} & \text{remove } \boxed{t}; & \top \\ 0.5 & \boxed{S}\boxed{t} & \text{remove } \boxed{t}; & \top \end{bmatrix}$$

$$\mathcal{RC}[\![\text{add } \boxed{r};]\!](\begin{array}{c}\boxed{t}\\ \boxed{S}\end{array}, \text{remove } \boxed{t};) \;=\; \begin{bmatrix} & \boxed{t} & & \\ 1.0 & \boxed{S} & \text{remove } \boxed{t}; & \bot \end{bmatrix}$$

$$\mathcal{RC}[\![\text{add } \boxed{r};]\!](\boxed{S}\boxed{t}, \text{remove } \boxed{t};) \;=\; \begin{bmatrix} 1.0 & \boxed{S}\boxed{t} & \text{remove } \boxed{t}; & \bot \end{bmatrix}$$

$$\texttt{tt path (3).}\quad \mathcal{RC}[\![RC_1\ RC_2]\!](\boxed{S}, \epsilon) \;=$$

$$\begin{bmatrix} 0.5 & \begin{bmatrix} & \boxed{t} & & \\ 1.0 & \boxed{S} & \text{remove } \boxed{t}; & \bot \end{bmatrix} & \top \\ 0.5 & \begin{bmatrix} 1.0 & \boxed{S}\boxed{t} & \text{remove } \boxed{t}; & \bot \end{bmatrix} & \top \end{bmatrix} \;=\; \begin{bmatrix} & \boxed{t} & & \\ 0.5 & \boxed{S} & \text{remove } \boxed{t}; & \bot \\ 0.5 & \boxed{S}\boxed{t} & \text{remove } \boxed{t}; & \bot \end{bmatrix}$$

The second path is a single reversible command.

$$\texttt{ff path.}\quad \mathcal{RC}[\![\text{add } \boxed{r};]\!](\boxed{S}, \epsilon) \;=\; \begin{bmatrix} 1.0 & \boxed{S} & \epsilon & \bot \end{bmatrix}$$

These paths are joined in $r(L)(\boxed{S})$.

$$\texttt{join.}\quad r(\{1,\texttt{tt},\text{add}\ \boxed{t};\ \text{add}\ \boxed{r};\},\{1,\texttt{ff},\text{add}\ \boxed{r};\})(\boxed{S}) \;=\; \begin{bmatrix} 0.5 & \texttt{tt} & \begin{bmatrix} 0.5 & \begin{array}{c}\boxed{t}\\ \boxed{S}\end{array} & \text{remove}\ \boxed{t}; & \bot \\ 0.5 & \boxed{S}\boxed{t} & \text{remove}\ \boxed{t}; & \bot \end{bmatrix} \\ 0.5 & \texttt{ff} & \begin{bmatrix} 1.0 & \boxed{S} & \epsilon & \bot \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 0.25 & \texttt{tt} & \begin{array}{c}\boxed{t}\\ \boxed{S}\end{array} & \text{remove}\ \boxed{t}; & \bot \\ 0.25 & \texttt{tt} & \boxed{S}\boxed{t} & \text{remove}\ \boxed{t}; & \bot \\ 0.5 & \texttt{ff} & \boxed{S} & \epsilon & \bot \end{bmatrix}$$

Lets move on to the calculation in the fixpoint function.

$$(\lambda h.\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L)))(\boxed{S}) \;=\; \lambda h.\ (retry(h)\ \circledcirc^{\mathcal{D}} \begin{bmatrix} 0.25 & \texttt{tt} & \begin{array}{c}\boxed{t}\\ \boxed{S}\end{array} & \text{remove}\ \boxed{t}; & \bot \\ 0.25 & \texttt{tt} & \boxed{S}\boxed{t} & \text{remove}\ \boxed{t}; & \bot \\ 0.5 & \texttt{ff} & \boxed{S} & \epsilon & \bot \end{bmatrix}) \;=$$

$$\lambda h.\ cont(h)\ \circledcirc^{\mathcal{D}} \begin{bmatrix} 0.25 & \begin{bmatrix} 1.0 & \texttt{tt} & \boxed{S} & \text{add}\ \boxed{t}; & \top \end{bmatrix} \\ 0.25 & \begin{bmatrix} 1.0 & \texttt{tt} & \boxed{S} & \text{add}\ \boxed{t}; & \top \end{bmatrix} \\ 0.5 & \begin{bmatrix} 1.0 & \texttt{ff} & \boxed{S} & \epsilon & \top \end{bmatrix} \end{bmatrix} = \lambda h.\ cont(h)\ \circledcirc^{\mathcal{D}} \begin{bmatrix} 0.5 & \texttt{tt} & \boxed{S} & \text{add}\ \boxed{t}; & \top \\ 0.5 & \texttt{ff} & \boxed{S} & \epsilon & \top \end{bmatrix} =$$

$$\lambda h.\ h(\boxed{S})$$

We can write $(\lambda h.\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L)))(\boxed{S}) = \lambda h.\ h(\boxed{S})$ as $\lambda h.\ \lambda G.\ (retry(h) \circledcirc^{\mathcal{D}} r(L)) = \lambda h.\ h$, which is the identity function. After reversing the failed executions, the only assemblies in the returned distribution were the initial distribution, making this a never terminating branch loop.

6 Conclusion

This thesis provides the formal semantics for the Algorithmic Language for Chemistry (ALCH), a high-level imperative programming language that allows programmers to define sequential Chemical Reaction Network-Controlled Tile Assembly Models in an efficient manner. The most challenging aspect of ALCH, in terms of defining its semantics, was its probabilistic nature and the branch statement. To define its operational semantics the sub-distribution monad was utilized. This monad allows the operational semantics to take a program state and return a distribution on program states. Proofs were provided that show that this distribution is a sub-probability distribution. The latter part of this thesis defines a denotational semantics using a number of categorical constructs.

The logical next step for this work is to formally prove correspondence between the operational and denotational semantics. If one would want to use the formal semantics to analyse and/or verify programs, it might prove useful to introduce a fail state to catch CRN-TAM commands that cannot execute. These may occur when an attempt is made to add or remove a tile from an assembly when that is not possible. In the operational semantics, there simply are no inference rules for these situations, meaning one would have subtract the sum of the probabilities of all elements in the sub-probability distribution from 1 to find the probability that a program crashes. By introducing a fail state and fail transitions this could be avoided.
The compositions in the denotational semantics are defined using in total seven monads, of which all but one are very similar to two others. Finding a more elegant solution to compose functions with a probability distribution in their co-domains would be a welcome alteration, especially since the denotational semantics are the least readable part of this project.

7 Acknowledgements

I want to thank my first supervisor, Henning Basold, for his time, help and knowledge of semantics, domain theory and category theory, as well as my second supervisor, Jetty Kleijn, for her knowledge on molecular computing. Additional thanks goes out to Rintse van de Vlasakker, who was eager to converse about every challenging aspect of this project. Furthermore, I would like to thank the creators of ALCH for kindly providing me information on the language that was not yet public.

References

[1] Titus H. Klinge, James I. Lathrop, Sonia Moreno, Hugh D. Potter, Narun K. Raman, and Matthew R. Riley. ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*, volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[2] Nicholas Schiefer and Erik Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In Andrew Phillips and Peng Yin, editors, *DNA Computing and Molecular Programming*, pages 34–54, Cham, 2015. Springer International Publishing.

[3] Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

[4] Matthew J. Patitz. An introduction to tile-based self-assembly. In *UCNC*, 2012.

[5] Roberto Bruni and Ugo Monatanari. *Models of Computation*. Springer, 2017.

[6] Martin Churchill. Abstract semantics for a quantum programming language. Master's thesis, University of Oxford, 2007.

[7] David Yu Zhang and Georg Seelig. Dynamic dna nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103—113, February 2011.

[8] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. In Darko Stefanovic and Andrew Turberfield, editors, *DNA Computing and Molecular Programming*, pages 25–42, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] James I. Lathrop, Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete sierpinski triangles. *Theoretical Computer Science*, 410(4):384–405, 2009. Computational Paradigms from Nature.

[10] I. Hasuo, B.P.F. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4):11–1/36, 2007.

[11] Harish Chandran, Nikhil Gopalkrishnan, and John Reif. The tile complexity of linear assemblies. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, pages 235–253, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] Marko Vasic, David Soloveichik, and Sarfraz Khurshid. CRN++: molecular programming language. *CoRR*, abs/1809.07430, 2018.

[13] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 81–94, New York, NY, USA, 1989. Association for Computing Machinery.

[14] Cedric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *IN PROCEEDINGS OF THE 23RD ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, POPL '96, pages 372–385. ACM Press, 1996.

[15] Louis Mandel and Luc Maranget. Programming in JoCaml — Extended Version. Research Report RR-6261, INRIA, 2007.

[16] Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. Oxford University Press, Inc., USA, 1995.

[17] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. *Science of Computer Programming*, 185, 2020.