

ALGEBRAIC INFORMATION EFFECTS

Chao-Hong Chen

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering,
Indiana University
August 2021

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Amr Sabry, PhD

Lawrence S. Moss, PhD

Yan Huang, PhD

Chung-chieh Shan, PhD

March 31,2021

Copyright © 2021

Chao-Hong Chen

Chao-Hong Chen

ALGEBRAIC INFORMATION EFFECTS

From the informational perspective, programs that are usually considered as pure have effects, for example, the simply typed lambda calculus is considered as a pure language. However, β -reduction does not preserve information and embodies information effects. To capture the idea about pure programs in the informational sense, a new model of computation — reversible computation was proposed. This work focuses on type-theoretic approaches for reversible effect handling. The main idea of this work is inspired by compact closed categories. Compact closed categories are categories equipped with a dual object for every object. They are well-established as models of linear logic, concurrency, and quantum computing. This work gives computational interpretations of compact closed categories for conventional product and sum types, where a negative type represents a computational effect that “reverses execution flow” and a fractional type represents a computational effect that “allocates/deallocates space”.

Amr Sabry, PhD

Lawrence S. Moss, PhD

Yan Huang, PhD

Chung-chieh Shan, PhD

CONTENTS

Abstract	iv
1 Introduction	1
2 Preliminaries	10
2.1 Reversible Machines	10
2.2 Introduction to Π	14
2.2.1 Syntax of Π	15
2.2.2 Abstract Machine Semantics of Π	17
2.2.3 Big-step Interpreter of Π	21
2.2.4 Rig groupoid	23
2.2.5 Examples	26
2.3 Introduction to Π^{++}	32
2.4 Space-time Trade-off	34
3 Negative Types	37
3.1 Syntax of Π^-	38
3.2 Abstract Machine Semantics of Π^-	38
3.3 Big-step Interpreter of Π^-	43
3.4 Compact Closed Category	48
3.5 Examples	53
4 Fractional Types	58
4.1 Syntax of $\Pi/$	59
4.2 Abstract Machine Semantics of $\Pi/$	60

4.3	Big-step Interpreter of Π'	63
4.4	Compact Closed Category	64
4.5	Examples	65
5	Field of Types	69
5.1	Syntax of Π^Q	69
5.2	Abstract Machine Semantics of Π^Q	69
5.3	Big-step Interpreter of Π^Q	74
5.4	Examples	78
6	Conclusion and Future Works	83
	Bibliography	85
	Curriculum Vitae	

CHAPTER 1

INTRODUCTION

The study of effects is an important topic in programming language research. One vital definition is whether a program is effectful or not. Many formal definitions have been proposed to address this problem [38]. However, even for simply typed lambda calculus (STLC) [16], which satisfies all of those definitions and is always considered as a pure language, it also contains *information effects* [27] that are usually overlooked.

Consider the following STLC program:

$$\lambda(x, y).x : (\mathbb{B} \times \mathbb{B}) \rightarrow \mathbb{B}$$

where $\mathbb{B} = \{\mathbb{F}, \mathbb{T}\}$. This program takes two bits and discards the second one. It satisfies all the conventional definitions of purity. However, from the informational perspective, this function's input has two bits of information but the output only has one, i.e. one bit of information is lost during the computation. In [27], James and Sabry defined the change of information during computations as *information effects*. To be more specific, for a function $f : A \rightarrow B$:

- The input entropy of f is the entropy of $a : A$ with given input distribution on A .
- The output entropy of f is the entropy of $f(a) : B$ given the input distribution on A .

The information effect of f is the difference between its input and output entropy, and f is reversible if its input entropy is the same as output entropy. To expose the hidden information effects in irreversible programs, James and Sabry defined an irreversible programming language **LET** and compiled it to a reversible programming language Π° [27]. The compiling technique is similar to Toffoli's idea [42]: introducing new inputs to make it

reversible. The number of new inputs introduced during compilation will correspond to the amount of information effects.

To capture pure computations in the informational sense, a new computational model – reversible computation was proposed. It seems that this model adds restrictions on the classical computational model so it is less powerful. But Toffoli [42] proved that reversible computation is universal:

Theorem 1.1. *For any finite function $\phi : \mathbb{B}^m \rightarrow \mathbb{B}^n$ there exists an reversible finite function $f : \mathbb{B}^r \times \mathbb{B}^m \rightarrow \mathbb{B}^n \times \mathbb{B}^{r+m-n}$, with $r \leq n$, such that*

$$f(x_1, \dots, x_m, \overbrace{\mathbb{F}, \dots, \mathbb{F}}^r) = (y_1, \dots, y_n, g_1, \dots, g_{r+m-n})$$

where $(y_1, \dots, y_n) = \phi(x_1, \dots, x_m)$ and (g_1, \dots, g_{r+m-n}) are arbitrary garbage outputs.

This theorem states that every irreversible Boolean function can be computed by using a reversible Boolean function with additional inputs/outputs. There are two main applications of reversible computation:

- Quantum computing: Since all the quantum logic gates are reversible, the design of quantum programming languages and reversible programming languages are highly related. A good reversible programming language could lead to a better quantum programming language design.
- Ultra-low-power chips: According to Landauer’s principle [33], any logically irreversible operation will necessarily dissipate heat. Hence, reversible computation provides a promising direction for ultra-low-power chip design [19, 20].

Since the proposal of reversible computation, multiple reversible programming languages were developed [40, 41, 44, 45]. However, they do not have type systems that can help programmers handle effects:

- rFun [44]: rFun allows expressions to freely allocate constant values:

```
data Bool = True | False

ex :: Bool <-> (Bool, Bool)
```

```
ex b = (b , False)
```

Using such expressions, it is possible to define expressions that behave like `qinit` and `qterm` in Quipper [25]:

```
initF :: () <-> Bool
initF () = False
```

```
termF :: Bool <-> ()
termF False = ()
```

However, the type system does not keep track of the effects of introducing ancilla values. So, the programmers need to keep track of these effects on their own. Moreover, it also accepts programs that do not preserve information:

```
term :: Bool <-> ()
term x = ()
```

- Ricercar [40]: Ricercar is a description language for reversible Boolean gates, which does not have a type system. It supports scoped ancilla wire through the expression $\alpha x.A(x)$, where gate A can use the wire x which is initialized to 0, and A needs to reset x back to 0 when the computation is finished.
- Janus [41, 45]: Janus is an imperative programming language that does not have a type system. Since all variables are global, every function is effectful.

To fill the missing pieces, this work will study effect handling in reversible programming languages using a type-theoretic approach.

The starting point of this work is the pure reversible programming Π [12], which has nice algebraic properties inherited from category theory. The type system of Π contains empty, unit, sum, and product types:

$$\frac{}{0 : \mathbb{U}} \quad \frac{}{1 : \mathbb{U}} \quad \frac{\tau_1 : \mathbb{U} \quad \tau_2 : \mathbb{U}}{\tau_1 + \tau_2 : \mathbb{U}} \quad \frac{\tau_1 : \mathbb{U} \quad \tau_2 : \mathbb{U}}{\tau_1 \times \tau_2 : \mathbb{U}}$$

The programs of Π are constructed using combinators, which represent type isomorphisms.

All the combinators are from the definition of bimonoidal category:

- From the definition of category [36]:

Definition 1.2. *A category C consists of:*

- a class $Obj(C)$ of objects
- a class $Hom(C)$ of morphisms between the objects
- composition of morphisms $_ \circ _$: for morphisms $f : a \rightarrow b$ and $g : b \rightarrow c$,
 $g \circ f : a \rightarrow c$.

which satisfy

- Identity: For every object x there exists a morphism 1_x such that:
 - * For all $f : a \rightarrow x$, $1_x \circ f = f$.
 - * For all $g : x \rightarrow b$, $g \circ 1_x = g$.
- Associativity: $(f \circ g) \circ h = f \circ (g \circ h)$.

Π contains combinators:

$$\begin{array}{c} \text{id} \leftrightarrow : \tau \leftrightarrow \tau : \text{id} \leftrightarrow \\ \frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_2 \leftrightarrow \tau_3}{c_1 \circ c_2 : \tau_1 \leftrightarrow \tau_3} \end{array}$$

$\text{id} \leftrightarrow$ has type $\tau \leftrightarrow \tau$, which corresponds to the identity morphism 1_τ for every object.

$_ \circ _$ corresponds to the morphism composition $_ \circ _$, but $_ \circ _$ represents sequential composition ($c_1 \circ c_2 = c_2 \circ c_1$).

- From the definition of monoidal category [37]:

Definition 1.3. *A monoid category is a category C equipped with*

- Tensor product: bifunctor $_ \otimes _ : C \times C \rightarrow C$
- Identity object: $I \in Obj(C)$
- Natural isomorphisms:
 - * $\alpha : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$
 - * $\lambda_A : I \otimes A \simeq A$

which satisfy the pentagon and triangle diagrams.

Π contains combinators for monoidal structure of sum types:

$$\begin{aligned}
\text{unite}_+ | & : \quad \mathbb{0} + \tau \leftrightarrow \tau & : \text{unit}_+ | \\
\text{assocl}_+ : & \tau_1 + (\tau_2 + \tau_3) \leftrightarrow (\tau_1 + \tau_2) + \tau_3 & : \text{assoc}_+ \\
& \frac{c_1 : \tau_1 \leftrightarrow \tau_3 \quad c_2 : \tau_2 \leftrightarrow \tau_4}{c_1 \oplus c_2 : \tau_1 + \tau_2 \leftrightarrow \tau_3 + \tau_4}
\end{aligned}$$

$\text{unite}_+ |$ and $\text{unit}_+ |$ correspond to the natural isomorphism λ . assocl_+ and assoc_+ correspond to the natural isomorphism α . And $_ \oplus _$ corresponds to the bifunctor on morphisms. Similarly for product types:

$$\begin{aligned}
\text{unite}_* | & : \quad \mathbb{1} \times \tau \leftrightarrow \tau & : \text{unit}_* | \\
\text{assocl}_* : & \tau_1 \times (\tau_2 \times \tau_3) \leftrightarrow (\tau_1 \times \tau_2) \times \tau_3 & : \text{assoc}_* \\
& \frac{c_1 : \tau_1 \leftrightarrow \tau_3 \quad c_2 : \tau_2 \leftrightarrow \tau_4}{c_1 \otimes c_2 : \tau_1 \times \tau_2 \leftrightarrow \tau_3 \times \tau_4}
\end{aligned}$$

- From the definition of symmetric monoidal category [36]:

Definition 1.4. *A symmetric monoid category is a monoidal category (C, \otimes, I) such that for every pair $A, B \in \text{Obj}(C)$ there is an isomorphism: $s_{A,B} : A \otimes B \simeq B \otimes A$ such that the unit, associativity and inverse coherence conditions hold.*

Π contains combinators for sum types:

$$\text{swap}_+ : \tau_1 + \tau_2 \leftrightarrow \tau_2 + \tau_1 \quad : \text{swap}_+$$

and for product types:

$$\text{swap}_* : \tau_1 \times \tau_2 \leftrightarrow \tau_2 \times \tau_1 \quad : \text{swap}_*$$

- From the definition of bimonoidal category [34]:

Definition 1.5. *A bimonoidal category C is a category with two symmetric monoidal structures $(C, \oplus, \mathbb{0})$ and $(C, \otimes, \mathbb{1})$, together with the natural isomorphisms:*

- *Distributivity:* $d : A \otimes (B \oplus C) \simeq (A \otimes B) \oplus (A \otimes C)$
- *Absorption:* $a : A \otimes \mathbb{0} \simeq \mathbb{0}$

Π contains combinators:

$$\begin{aligned} \text{absorbr} : \quad & \mathbb{0} \times \tau \leftrightarrow \mathbb{0} & : \text{factorzl} \\ \text{dist} : \quad & (\tau_1 + \tau_2) \times \tau_3 \leftrightarrow (\tau_1 \times \tau_3) + (\tau_2 \times \tau_3) & : \text{factor} \end{aligned}$$

The detailed syntax and operational semantics of Π is given in Sec. 2.2.

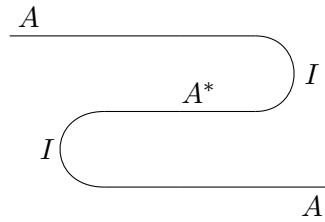
The goal of this work is to introduce effects to Π and to extend Π 's type system to track effects. To achieve this, we take the inspiration from compact closed categories [31, 32] which provide nice algebraic structures:

Definition 1.6. *A compact closed category is a symmetric monoidal category (C, \otimes, I) such that*

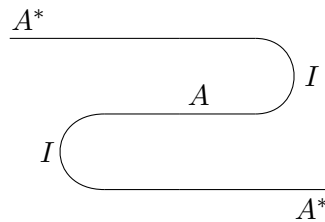
- Every object $A \in \text{Obj}(C)$ has dual object A^* .
- For all $A \in \text{Obj}(C)$ there exists unit $\eta_A : I \rightarrow A^* \otimes A$ and counit $\epsilon_A : A \otimes A^* \rightarrow I$.

which satisfy the snake diagrams:

- $A \xrightarrow{\cong} A \otimes I \xrightarrow{A \otimes \eta} A \otimes (A^* \otimes A) \xrightarrow{\cong} (A \otimes A^*) \otimes A \xrightarrow{\epsilon \otimes A} I \otimes A \xrightarrow{\cong} A = id_A$



- $A^* \xrightarrow{\cong} I \otimes A^* \xrightarrow{\eta \otimes A^*} (A^* \otimes A) \otimes A^* \xrightarrow{\cong} A^* \otimes (A \otimes A^*) \xrightarrow{A^* \otimes \epsilon} A^* \otimes I \xrightarrow{\cong} A^* = id_{A^*}$



Compact closed categories have numerous applications in mathematics, physics, and computer science. Examples of compact closed categories include models of multiplicative linear logic [5], models of concurrency [3], and models of quantum computing [1, 22]. Compact closed categories are a natural extension of symmetric monoidal categories, which have nice algebraic properties. We will extend the two monoidal structures of Π to compact closed

categories. For the additive monoidal structure, we will introduce negative types where a negative type represents the effect on time that “reverses execution flow”. The extended language Π^- will be introduced in Chap. 3. For the multiplicative monoidal structure, we will introduce fractional types where a fractional type represents the effect on space that “allocates/deallocates space”. The extended language $\Pi^/$ will be introduced in Chap. 4. In the end, we will put everything together to form a language $\Pi^{\mathbb{Q}}$ which contains both negative and fractional types. To demonstrate its expressiveness, we will implement a SAT solver in $\Pi^{\mathbb{Q}}$ using the newly introduced effects.

The space and time effects inspired by compact closed categories are important for reversible computation:

- Space effects: Toffoli [42] showed that every reversible function can be obtained using only generalized AND/NAND functions, where the generalized AND/NAND function of order n (the number of input is n) is defined as $\theta_n(x_1, \dots, x_n) = (x_1, \dots, x_n \text{ xor } (x_1 \wedge \dots \wedge x_{n-1}))$:

Theorem 1.7. *Any reversible finite function of order n can be obtained by composition of generalized AND/NAND functions of order $\leq n$.*

This theorem implies that $\{\theta_n | n \geq 1\}$ is functionally complete for reversible computation. However this is an infinite collection of functions, is it possible to have a finite set of primitive functions that is complete for reversible computation like $\{\text{AND}, \text{NOT}\}$ for classical computation? Taking the idea from classical computation, because an n -bit AND gate could be obtained using normal 2-bit AND gates so we do not need the infinite collection of n -bit AND gates to form a complete set of primitive functions for classical computation. Could we do the same for reversible computation? Unfortunately, the answer is negative. Toffoli showed that there exist reversible functions that cannot be computed using smaller generalized AND/NAND functions:

Theorem 1.8. *There exist reversible finite functions of order n which cannot be obtained by composition of generalized AND/NAND functions of order strictly less than n .*

This theorem implies that we cannot obtain θ_n using $\{\theta_k \mid 1 \leq k < n\}$. To remedy this, Toffoli also showed that:

Theorem 1.9. *Any reversible function can be realized, possibly with temporary storage (but with no garbage!) by means of a reversible computational network using as primitives the generalized AND/NAND elements of order ≤ 3 .*

This theorem implies that with the space effects that allowed us to allocate/deallocate temporary storage $\{\theta_1, \theta_2, \theta_3\}$ forms a complete set of primitive gates for reversible computation. Hence space effects are very important for reversible computation, and they appear in every reversible programming language.

- Time effects: The ability to change execution flows is vital for almost every programming language, no matter if it is reversible or not. Assume that we want to write a higher-order function $iter : Int \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A$ such that $iter(n)(f)(a) = \overbrace{f(\dots f(a)\dots)}^n$. Without the ability to change execution flow the only option to write this function is to write down n number of calls to f in the program:

$$\begin{aligned} iter(0)(f)(a) &= a \\ iter(1)(f)(a) &= f(a) \\ iter(2)(f)(a) &= f(f(a)) \\ iter(3)(f)(a) &= f(f(f(a))) \\ &\vdots \end{aligned}$$

Assume that we use 32-bit Int then this program will have 2^{32} calls to f , which results in an exponential sized program. Hence to make the programming language practical, it is very important to have the ability to change execution flow so that code can be reused at runtime. For example, the $iter$ function can be easily implemented using recursion:

$$\begin{aligned} iter(0)(f)(a) &= a \\ iter(n+1)(f)(a) &= iter(n)(f)(f(a)) \end{aligned}$$

or loop:

```
ans = a;  
for (i=0; i<n; i=i + 1)ans = f(ans);  
return ans;
```

Note that in both programs, there is only one call to f . As a result, both programs have constant sizes. These effects are very important in practice, especially when using programming languages that are very closed to circuits like Π . Without the ability to construct feedback circuits, several function implementations will have exponential sized circuits.

With the above background, I can state my thesis:

The duals of algebraic data types represent useful reversible effects.

The rest of this dissertation is structured as follows:

- Chapter 2 will provide a detailed introduction to Π , formalization of reversible machines, and some background knowledge.
- Chapter 3 will describe the extension of negative types to Π , and its compact closed category construction.
- Chapter 4 will describe the extension of fractional types to Π , and its compact closed category construction.
- Chapter 5 will put fractional and negative types together and concludes this work.

Most of the theorems are formalized in Agda, which will be marked using $\mathcal{U}Agda$.

PUBLICATIONS This dissertation’s main results have been published in:

- Chao-Hong Chen and Amr Sabry. A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types. In *Symposium on Principles of Programming Languages 2021 (POPL’21)* [15].
- Chao-Hong Chen, Vikraman Choudhury, Jacques Carette, Amr Sabry. Fractional Types: Expressive and Safe Space Management for Ancilla Bits. In *Proceedings of the 12th International Conference on Reversible Computation (RC’20)* [14].

CHAPTER 2

PRELIMINARIES

This chapter will provide preliminaries, which will be used throughout the whole dissertation.

Three main topics will be introduced:

- Reversible machines: A formal definition of reversible machines is revisited, and some meta-theoretical properties will be discussed. All the machine semantics given in this dissertation are reversible, hence it is vital to base it on a formal definition of reversible machines that allows rigorous reasoning and discussion about its meta-theoretical properties.
- Reversible programming language Π : A detailed introduction of Π 's syntax, machine semantics, and a big-step interpreter will be given. And various meta-theoretical properties will be proved. Based on this, various extensions will be proposed in the following chapters.
- Reversible programming language Π^{++} : Π^{++} provides higher-level Haskell-like syntax over Π , which makes the presentation of programs easier to understand.
- Space-time trade-offs: An small example is given, which demonstrates the space-time trade-offs of product and sum types. This example will develop the intuition about the correspondence between space/time and product/sum types.

2.1 REVERSIBLE MACHINES

In this section, the formal definition of reversible abstract machines is given and discussed.

First, starting with the formal definition of abstract machines:

Definition 2.1 (Abstract Machine). *An abstract machine is a pair (S, \rightsquigarrow) , where S is the set of machine states and $\rightsquigarrow \subseteq S \times S$ is a binary relation on S which describes the transition between machine states.*

Normally a computation starts from an initial state and ends in a stuck state from which no more transition step can be taken, to be precise:

Definition 2.2 (Initial state). *$s \in S$ is an initial state of a machine (S, \rightsquigarrow) if there does not exist s' such that $s' \rightsquigarrow s$.*

Definition 2.3 (Stuck state). *$s \in S$ is a stuck state of a machine (S, \rightsquigarrow) if there does not exist s' such that $s \rightsquigarrow s'$.*

Based on Definition 2.1, reversible abstract machines can be defined:

Definition 2.4 (Reversible Abstract Machine). *A reversible abstract machine is an abstract machine (S, \rightsquigarrow) such that*

- *\rightsquigarrow is forward deterministic: $\forall s, s_1, s_2 \in S$ if $s \rightsquigarrow s_1$ and $s \rightsquigarrow s_2$ then $s_1 = s_2$.*
- *\rightsquigarrow is backward deterministic: $\forall s, s_1, s_2 \in S$ if $s_1 \rightsquigarrow s$ and $s_2 \rightsquigarrow s$ then $s_1 = s_2$.*

Remark. *This is arguably the most general definition of reversible machines. For example, the biorthogonal pattern-matching automaton [2] is a special case of reversible abstract machines where the transition relations are restricted to states carrying unifiable terms.*

Some easy lemmas can be proved about reversible abstract machines. First, let's define some convenient notations:

Definition 2.5. *For any binary relation $R \subseteq S \times S$:*

- *Let R^* be the reflexive and transitive relation of R :*

$$\frac{}{sR^*s} \quad \frac{sRs' \quad s'R^*s''}{sR^*s''}$$

- *Let R^\dagger be the inverse relation such that $s'R^\dagger s$ iff sRs' .*
- *Let R^n be the n -step transitive relation:*

$$\frac{}{sR^0s} \quad \frac{sR^n s' \quad s'R s''}{sR^{n+1} s''}$$

Using these notations, some properties of reversible abstract machines can be proved:

Lemma 2.6. $\mathcal{U}Agda$ For any reversible abstract machine (S, \rightsquigarrow) , any $n \in \mathbb{N}$ and $s_0, s_n, s'_n \in S$, if $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^n s'_n$ then $s_n = s'_n$.

Proof. By induction on n :

- $n = 0$: $s_0 = s_n = s'_n$.
- $n = m + 1$: if $s_0 \rightsquigarrow^m s_m \rightsquigarrow s_{m+1}$ and $s_0 \rightsquigarrow^m s'_m \rightsquigarrow s'_{m+1}$, by induction hypothesis we have $s_m = s'_m$. Since \rightsquigarrow is forward deterministic so $s_{m+1} = s'_{m+1}$.

□

Theorem 2.7. $\mathcal{U}Agda$ For any reversible abstract machine (S, \rightsquigarrow) and $s_0, s, s' \in S$, if $s_0 \rightsquigarrow^* s$, $s_0 \rightsquigarrow^* s'$, and both s and s' are stuck states then $s = s'$.

Proof. Since $s_0 \rightsquigarrow^* s$ and $s_0 \rightsquigarrow^* s'$, so there exist $n, m \in \mathbb{N}$ such that $s_0 \rightsquigarrow^n s$ and $s_0 \rightsquigarrow^m s'$. If $n \neq m$, without loss of generality, assume $n < m$ by Lemma 2.6 $s_0 \rightsquigarrow^n s \rightsquigarrow^{m-n} s'$ which contradicts the assumption that s is stuck. So it must be the case that $n = m$, and by Lemma 2.6 $s = s'$.

□

Similar properties can also be proved for \rightsquigarrow^\dagger :

Lemma 2.8. $\mathcal{U}Agda$ For any reversible abstract machine (S, \rightsquigarrow) , any $n \in \mathbb{N}$ and $s_0, s_n, s'_n \in S$, if $s_0 \rightsquigarrow^{\dagger n} s_n$ and $s_0 \rightsquigarrow^{\dagger n} s'_n$ then $s_n = s'_n$.

Proof. Similar to Lemma 2.6.

□

Theorem 2.9. $\mathcal{U}Agda$ For any reversible abstract machine (S, \rightsquigarrow) and $s_0, s, s' \in S$, if $s_0 \rightsquigarrow^{\dagger*} s$, $s_0 \rightsquigarrow^{\dagger*} s'$, and both s and s' are initial states then $s = s'$.

Proof. Similar to Theorem 2.7.

□

A remarkable property of reversible abstract machines is that starting from initial states no state will ever repeat during an execution.

Theorem 2.10. $\mathcal{U}Agda$ For any reversible abstract machine (S, \rightsquigarrow) , if $s_0 \in S$ is an initial state then for any $n < m$ and $s_n, s_m \in S$ such that $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^m s_m$ we have that $s_n \neq s_m$.

Proof. If $s_n = s_m$, since $s_0 \rightsquigarrow^n s_n$, $s_0 \rightsquigarrow^m s_m$ and $n < m$ so $s_n \rightsquigarrow^{\dagger n} s_0$ and $s_n \rightsquigarrow^{\dagger n} s_{m-n}$. By Lemma 2.8, $s_0 = s_{m-n}$. However, since $n < m$ there must exist s_{m-n-1} such that $s_{m-n-1} \rightsquigarrow s_{m-n} = s_0$ which contradicts the assumption that s_0 is an initial state. Hence, $s_n \neq s_m$. \square

This theorem will be very useful when proving termination for a reversible abstract machine that has only a finite number of reachable states from every initial state.

The reversible abstract machines defined in Definition 2.1 compute total reversible functions. A more general class of machines that compute partial reversible functions are partial reversible abstract machines:

Definition 2.11 (Partial Reversible Abstract Machine). *A partial reversible machine is a triple $(S, \mathcal{E}, \rightsquigarrow)$, where S is the set of machine states, $\mathcal{E} \subseteq S$ is a set of error states, and \rightsquigarrow is a transition relation satisfying the following three properties:*

- \rightsquigarrow is forward deterministic: $\forall s, s_1, s_2 \in S$ if $s \rightsquigarrow s_1$ and $s \rightsquigarrow s_2$ then $s_1 = s_2$.
- \rightsquigarrow is backward deterministic on proper states: $\forall s, s_1, s_2 \in S$ if $s \notin \mathcal{E}$, $s_1 \rightsquigarrow s$, and $s_2 \rightsquigarrow s$ then $s_1 = s_2$.
- If $s \in \mathcal{E}$ then s is stuck.

Similar properties can also be proved for partial reversible machines with slight modifications:

Lemma 2.12. $\mathcal{U}Agda$ *For any partial reversible abstract machine $(S, \mathcal{E}, \rightsquigarrow)$, any $n \in \mathbb{N}$ and $s_0, s_n, s'_n \in S$, if $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^n s'_n$ then $s_n = s'_n$.*

Proof. Similar to Lemma 2.6. \square

Theorem 2.13. $\mathcal{U}Agda$ *For any reversible abstract machine $(S, \mathcal{E}, \rightsquigarrow)$ and $s_0, s, s' \in S$, if $s_0 \rightsquigarrow^* s$, $s_0 \rightsquigarrow^* s'$ and both s and s' are stuck states then $s = s'$.*

Proof. Similar to Lemma 2.7. \square

Lemma 2.14. $\mathcal{U}Agda$ *For any reversible abstract machine $(S, \mathcal{E}, \rightsquigarrow)$, any $n \in \mathbb{N}$ and $s_0, s_n, s'_n \in S$, if $s_0 \notin \mathcal{E}$, $s_0 \rightsquigarrow^{\dagger n} s_n$ and $s_0 \rightsquigarrow^{\dagger n} s'_n$ then $s_n = s'_n$.*

Proof. Similar to Lemma 2.8. □

Theorem 2.15. $\mathcal{U}^{\text{Agda}}$ For any reversible abstract machine $(S, \mathcal{E}, \rightsquigarrow)$ and $s_0, s, s' \in S$, if $s_0 \notin \mathcal{E}$, $s_0 \rightsquigarrow^{\dagger^*} s$, $s_0 \rightsquigarrow^{\dagger^*} s'$ and both s and s' are initial states then $s = s'$.

Proof. Similar to Theorem 2.9. □

And the non-repeating property still holds for partial reversible abstract machines since failure states are stuck:

Theorem 2.16. $\mathcal{U}^{\text{Agda}}$ For any partial reversible machine $(S, \mathcal{E}, \rightsquigarrow)$, if $s_0 \in S$ is an initial state then for any $n < m$ and $s_n, s_m \in S$ such that $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^m s_m$ we have that $s_n \neq s_m$.

Proof. Assume $s_n = s_m$, if $s_n \in \mathcal{E}$ then it is stuck and there does not exist s_m such that $s_0 \mapsto^n s_n \mapsto^{m-n} s_m$. Hence $s_m = s_n \notin \mathcal{E}$. Similar to Theorem 2.10. □

All machines that appear in this dissertation are either reversible or partial reversible machines.

2.2 INTRODUCTION TO Π

Π is a reversible programming language developed by Roshan and Sabry [27]. It originates from another reversible programming language Π^o [27], Carette and Sabry removed recursive types and added combinators inspired by rig categories [30,35]. As a result, Π is a reversible programming language that gives a computational interpretation to a rig category.

2.2.1 SYNTAX OF Π

The syntax of Π consists of several sorts:

<i>Value types</i>	$t ::= \mathbb{0} \mid \mathbb{1} \mid t + t \mid t \times t$
<i>Values</i>	$v ::= \mathbf{tt} \mid \mathbf{inj}_1 v \mid \mathbf{inj}_2 v \mid (v, v)$
<i>Combinator types</i>	$t \leftrightarrow t$
<i>Combinators</i>	$c ::= \mathbf{unite}_+ \mid \mathbf{uniti}_+ \mid \mathbf{swap}_+ \mid \mathbf{assocl}_+ \mid \mathbf{assocr}_+$ $\quad \mid \mathbf{unite}_* \mid \mathbf{uniti}_* \mid \mathbf{swap}_* \mid \mathbf{assocl}_* \mid \mathbf{assocr}_*$ $\quad \mid \mathbf{absorbr} \mid \mathbf{factorzl} \mid \mathbf{dist} \mid \mathbf{factor} \mid \mathbf{id} \leftrightarrow \mid c \mathbin{\&} c \mid c \oplus c \mid c \otimes c$
<i>Programs</i>	$p ::= c v$

Focusing on finite types, the building blocks of type theory are: the empty type ($\mathbb{0}$), the unit type ($\mathbb{1}$), the sum type ($+$), and the product (\times) type. The typing judgments are given in figure 2.1. One may view each type t as a collection of physical wires that can transmit $|t|$ distinct values where $|t|$ is a natural number that indicates the size of a type, computed as:

$$\begin{aligned}
 |\mathbb{0}| &= 0 \\
 |\mathbb{1}| &= 1 \\
 |t_1 + t_2| &= |t_1| + |t_2| \\
 |t_1 \times t_2| &= |t_1| * |t_2|
 \end{aligned}$$

Thus the type $\mathbb{B} = \mathbb{1} + \mathbb{1}$ corresponds to a wire that can transmit one of two values, i.e., bits, with the convention that $\mathbf{inj}_1 \mathbf{tt}$ represents \mathbb{F} and $\mathbf{inj}_2 \mathbf{tt}$ represents \mathbb{T} . The type $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ corresponds to a collection of wires that can transmit three bits. From that perspective, a type isomorphism between types t_1 and t_2 (such that $|t_1| = |t_2| = n$) models a *reversible* combinational circuit that *permutes* the n different values. These type isomorphisms are represented by the combinators of Π . A combinator c of type $t_1 \leftrightarrow t_2$ represents a type isomorphism between t_1 and t_2 . All the typing judgments for combinators are collected in Fig. 2.2.

$$\frac{}{\mathbf{tt} : \mathbb{1}} \quad \frac{v_1 : t_1 \quad v_2 : t_2}{(v_1, v_2) : t_1 \times t_2} \quad \frac{v_1 : t_1}{\mathbf{inj}_1 v_1 : t_1 + t_2} \quad \frac{v_2 : t_2}{\mathbf{inj}_2 v_2 : t_1 + t_2}$$

Figure 2.1: Typing judgments of Π values.

$$\begin{array}{lll} \mathbf{id} \leftrightarrow : & t \leftrightarrow t & : \mathbf{id} \leftrightarrow \\ \\ \mathbf{unite}_+ \mathbf{l} : & \mathbb{0} + t \leftrightarrow t & : \mathbf{uniti}_+ \mathbf{l} \\ \mathbf{swap}_+ : & t_1 + t_2 \leftrightarrow t_2 + t_1 & : \mathbf{swap}_+ \\ \mathbf{assocl}_+ : & t_1 + (t_2 + t_3) \leftrightarrow (t_1 + t_2) + t_3 & : \mathbf{assocr}_+ \\ \\ \mathbf{unite}_* \mathbf{l} : & \mathbb{1} \times t \leftrightarrow t & : \mathbf{uniti}_* \mathbf{l} \\ \mathbf{swap}_* : & t_1 \times t_2 \leftrightarrow t_2 \times t_1 & : \mathbf{swap}_* \\ \mathbf{assocl}_* : & t_1 \times (t_2 \times t_3) \leftrightarrow (t_1 \times t_2) \times t_3 & : \mathbf{assocr}_* \\ \\ \mathbf{absorbr} : & \mathbb{0} \times t \leftrightarrow \mathbb{0} & : \mathbf{factorzl} \\ \mathbf{dist} : & (t_1 + t_2) \times t_3 \leftrightarrow (t_1 \times t_3) + (t_2 \times t_3) & : \mathbf{factor} \\ \\ \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_2 \leftrightarrow t_3}{\vdash c_1 \mathbin{\&}; c_2 : t_1 \leftrightarrow t_3} & \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \oplus c_2 : t_1 + t_3 \leftrightarrow t_2 + t_4} & \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \otimes c_2 : t_1 \times t_3 \leftrightarrow t_2 \times t_4} \end{array}$$

Figure 2.2: Typing judgments of Π combinators.

Each line in the top part of the above table introduces a pair of dual constants that witness the type isomorphism in the middle. These are the *base* combinators of Π . Note how the above has two readings: first as a set of typing relations for a set of constants. Second, if these axioms are seen as universally quantified, orientable statements, they also induce transformations of the (traditional) values. The (categorical) intuition here is that these axioms have computational content because they witness isomorphisms rather than merely stating extensional equalities. The isomorphisms are extended to form a congruence relation by adding the three *congruence* constructors at the bottom of the figure that witness equivalence and compatible closure.

For convenience, eight additional combinators are introduced:

$$\begin{array}{lll} \mathbf{unite}_+ \mathbf{r} : & t + \mathbb{0} \leftrightarrow t & : \mathbf{uniti}_+ \mathbf{r} \\ \mathbf{unite}_* \mathbf{r} : & t \times \mathbb{1} \leftrightarrow t & : \mathbf{uniti}_* \mathbf{r} \\ \mathbf{absorbl} : & t \times \mathbb{0} \leftrightarrow \mathbb{0} & : \mathbf{factorzr} \\ \mathbf{distl} : & t_1 \times (t_2 + t_3) \leftrightarrow (t_1 \times t_2) + (t_1 \times t_3) & : \mathbf{factorl} \end{array}$$

They are all definable in Π so there is no need to extend Π with more base combinators:

$\text{unite}_{+r} = \text{swap}_{+} \circledast \text{unite}_{+l}$
 $\text{uniti}_{+r} = \text{uniti}_{+l} \circledast \text{swap}_{+}$
 $\text{unite}_{*r} = \text{swap}_{*} \circledast \text{unite}_{*l}$
 $\text{uniti}_{*r} = \text{uniti}_{*l} \circledast \text{swap}_{*}$
 $\text{absorbl} = \text{swap}_{*} \circledast \text{absorbr}$
 $\text{factorzr} = \text{factorzl} \circledast \text{swap}_{*}$
 $\text{distl} = \text{swap}_{*} \circledast \text{dist} \circledast (\text{swap}_{*} \oplus \text{swap}_{*})$
 $\text{factorl} = (\text{swap}_{*} \oplus \text{swap}_{*}) \circledast \text{factor} \circledast \text{swap}_{*}$

A well-formed Π program $(c \ v) : t_2$ will apply a combinator $c : t_1 \leftrightarrow t_2$ to $v_1 : t_1$ and results in a value $v_2 : t_2$. The typing judgment of Π programs is:

$$\frac{c : t_1 \leftrightarrow t_2 \quad v : t_1}{(c \ v) : t_2}$$

The details of Π 's operational semantics will be discussed in section 2.2.2 and 2.2.3.

It is known that these type isomorphisms are sound and complete for all permutations on finite types [21, 23] and hence that they are *complete* for expressing combinational circuits [24, 27, 42]. Algebraically, these types and combinators form a *commutative semiring* (up to type isomorphism). Logically they form a superstructural logic capturing space-time trade-offs [39]. Categorically, they form a *weak rig groupoid* [12] which will be proved in Section 2.2.4 after giving the operational semantics of Π . Moreover, they have a sound and complete model in homotopy type theory [11].

2.2.2 ABSTRACT MACHINE SEMANTICS OF Π

This section will give a detailed description of Π 's abstract machine semantics along with theorems about its properties. First, the semantics of the base combinators are specified in Fig. 2.3 using a function δ . It is clear that this function computes a type isomorphism for a given base combinator, hence it has an inverse δ^\dagger such that $\delta^\dagger(c, \delta(c, v)) = v$.

To specify the semantics of full programs, we use an abstract machine consisting of three registers: a code register containing a combinator c , a value register containing a value v ,

$$\begin{array}{ll}
\delta(\text{unite}_+, \text{inj}_2 v) = v & \delta(\text{uniti}_+, v) = \text{inj}_2 v \\
\delta(\text{swap}_+, \text{inj}_1 v) = \text{inj}_2 v & \\
\delta(\text{swap}_+, \text{inj}_2 v) = \text{inj}_1 v & \\
\delta(\text{assocl}_+, \text{inj}_1 v) = \text{inj}_1 (\text{inj}_1 v) & \delta(\text{assocr}_+, \text{inj}_1 (\text{inj}_1 v)) = \text{inj}_1 v \\
\delta(\text{assocl}_+, \text{inj}_2 (\text{inj}_1 v)) = \text{inj}_1 (\text{inj}_2 v) & \delta(\text{assocr}_+, \text{inj}_1 (\text{inj}_2 v)) = \text{inj}_2 (\text{inj}_1 v) \\
\delta(\text{assocl}_+, \text{inj}_2 (\text{inj}_2 v)) = \text{inj}_2 v & \delta(\text{assocr}_+, \text{inj}_2 v) = \text{inj}_2 (\text{inj}_2 v) \\
\delta(\text{unite}_*, (\text{tt}, v)) = v & \delta(\text{uniti}_*, v) = (\text{tt}, v) \\
\delta(\text{swap}_*, (x, y)) = (y, x) & \\
\delta(\text{assocl}_*, (x, (y, z))) = ((x, y), z) & \delta(\text{assocr}_*, ((x, y), z)) = (x, (y, z)) \\
\delta(\text{dist}, (\text{inj}_1 x, z)) = \text{inj}_1 (x, z) & \delta(\text{factor}, \text{inj}_1 (x, z)) = (\text{inj}_1 x, z) \\
\delta(\text{dist}, (\text{inj}_2 y, z)) = \text{inj}_2 (y, z) & \delta(\text{factor}, \text{inj}_2 (y, z)) = (\text{inj}_2 y, z)
\end{array}$$

Figure 2.3: Semantics of base combinators

and a continuation register containing a continuation κ . Continuations are lists of frames $F_1 \bullet F_2 \bullet \dots \bullet F_j \bullet \square$ where each frame has a “hole” representing a missing combinator and where holes are filled according to the order $F_j[\dots [F_2[F_1]] \dots]$. In other words, the missing combinator is used to fill the hole in F_1 and the result is used to fill the hole in F_2 and so on. Fig. 2.4 gives the definition of well-formed continuations where $\kappa : \text{CONT}_{A \leftrightarrow B}$ represents an evaluation context with a hole of type $A \leftrightarrow B$.

$$\begin{array}{c}
\frac{}{\square : \text{CONT}_{A \leftrightarrow B}} \quad \frac{c_2 : B \leftrightarrow C \quad \kappa : \text{CONT}_{A \leftrightarrow C}}{(\square \circledast c_2) \bullet \kappa : \text{CONT}_{A \leftrightarrow B}} \quad \frac{c_1 : A \leftrightarrow B \quad \kappa : \text{CONT}_{A \leftrightarrow C}}{(c_1 \circledast \square) \bullet \kappa : \text{CONT}_{B \leftrightarrow C}} \\
\\
\frac{c_2 : C \leftrightarrow D \quad \kappa : \text{CONT}_{A+C \leftrightarrow B+D}}{(\square \oplus c_2) \bullet \kappa : \text{CONT}_{A \leftrightarrow B}} \quad \frac{c_1 : A \leftrightarrow B \quad \kappa : \text{CONT}_{A+C \leftrightarrow B+D}}{(c_1 \oplus \square) \bullet \kappa : \text{CONT}_{C \leftrightarrow D}} \\
\\
\frac{c_2 : C \leftrightarrow D \quad y : C \quad \kappa : \text{CONT}_{A \times C \leftrightarrow B \times D}}{(\square \oplus [c_2, y]) \bullet \kappa : \text{CONT}_{A \leftrightarrow B}} \quad \frac{c_1 : A \leftrightarrow B \quad x : B \quad \kappa : \text{CONT}_{A \times C \leftrightarrow B \times D}}{([c_1, x] \oplus \square) \bullet \kappa : \text{CONT}_{C \leftrightarrow D}}
\end{array}$$

Figure 2.4: Well-formed Continuation Stacks

There are two kinds of machine states $\langle c \mid v \mid \kappa \rangle$ and $[c \mid v \mid \kappa]$. The first is an “enter” state where the focus is on the combinator c and the second is a “return” state where the focus is on the continuation κ . In more detail, in a state $\langle c \mid v \mid \kappa \rangle$ evaluation is about to apply c to v ; in a state $[c \mid v \mid \kappa]$ evaluation has just finished applying c resulting in v which is ready to be consumed by the continuation. These distinctions are made precise in the following definition.

Definition 2.17 (Π -machine states). A Π -machine state σ is either:

- An enter state: $\langle c \mid v \mid \kappa \rangle$ where $c : A \leftrightarrow B$, $v : A$, and $\kappa : \text{CONT}_{A \leftrightarrow B}$.
- A return state: $[c \mid v \mid \kappa]$ where $c : A \leftrightarrow B$, $v : B$, and $\kappa : \text{CONT}_{A \leftrightarrow B}$.

Fig. 2.5 gives the transition rules of the abstract machine. The first group defines transition steps starting from enter states: these evaluation steps either immediately return if the computation is trivial (cases \mapsto_1 and \mapsto_2) or push a frame onto the continuation stack to focus on the sub-combinator. The second group defines the transition steps from return states. In the first rule \mapsto_7 , evaluation of c_1 has just finished; as the first frame on the continuation stack is $(\square \circledast c_2)$, evaluation proceeds by entering c_2 . Rule \mapsto_{10} shows what happens when the evaluation of c_2 terminates. In that case, the evaluation of the sequence $c_1 \circledast c_2$ is complete and v is ready to be consumed by κ .

$$\begin{array}{lcl}
\langle c \mid v \mid \kappa \rangle & \mapsto_1 & [c \mid \delta(c, v) \mid \kappa] \quad \text{for base combinators } c \\
\langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle & \mapsto_2 & [\text{id} \leftrightarrow \mid v \mid \kappa] \\
\langle c_1 \circledast c_2 \mid v \mid \kappa \rangle & \mapsto_3 & \langle c_1 \mid v \mid (\square \circledast c_2) \bullet \kappa \rangle \\
\langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle & \mapsto_4 & \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle \\
\langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle & \mapsto_5 & \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle \\
\langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle & \mapsto_6 & \langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle \\
\\
[c_1 \mid v \mid (\square \circledast c_2) \bullet \kappa] & \mapsto_7 & \langle c_2 \mid v \mid (c_1 \circledast \square) \bullet \kappa \rangle \\
[c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa] & \mapsto_8 & \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle \\
[c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa] & \mapsto_9 & [c_1 \otimes c_2 \mid (x, y) \mid \kappa] \\
[c_2 \mid v \mid (c_1 \circledast \square) \bullet \kappa] & \mapsto_{10} & [c_1 \circledast c_2 \mid v \mid \kappa] \\
[c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa] & \mapsto_{11} & [c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa] \\
[c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa] & \mapsto_{12} & [c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa]
\end{array}$$

Figure 2.5: Π -abstract machine

The machine transition relation is both forward and backward deterministic:

Lemma 2.18 (Π -forward deterministic). $\mathcal{U}Agda$ If $\sigma \mapsto \sigma_1$ and $\sigma \mapsto \sigma_2$ then $\sigma_1 = \sigma_2$

Proof. By checking all cases of the transition relation. □

Lemma 2.19 (Π -backward deterministic). $\mathcal{U}Agda$ If $\sigma_1 \mapsto \sigma$ and $\sigma_2 \mapsto \sigma$ then $\sigma_1 = \sigma_2$

Proof. By checking all cases of the transition relation. □

Hence the machine is a reversible abstract machine. The evaluation of a Π program $(c\ v)$ will start from the machine state $\langle c \mid v \mid \square \rangle$ and the result will be extracted when it enters a stuck (final) state $[c \mid v' \mid \square]$.

Definition 2.20 (Π -forward evaluation). *The abstract machine induces an evaluation function $eval(c) : A \rightarrow B$ where $eval(c)(v_1) = v_2$ if $\langle c \mid v_1 \mid \square \rangle \mapsto^* [c \mid v_2 \mid \square]$.*

The function $eval(c)$ is a total function for all c . To see this, an analysis of stuck states is needed.

Lemma 2.21 (Π -stuck states). *\mathcal{U}_{Agda} For all states σ , if σ is stuck then $\sigma = [c \mid v \mid \square]$.*

Proof. By case analysis on all possible states. □

Using this lemma and the non-repeating theorem for reversible abstract machines, the totality of $eval(c)$ can be proved.

Theorem 2.22 (Π -termination). *\mathcal{U}_{Agda}^1 For all Π -combinators $c : A \leftrightarrow B$ and $v_1 : A$ there exists $v_2 : B$ such that $eval(c, v_1) = v_2$*

Proof. The set of reachable states starting from $\langle c \mid v_1 \mid \square \rangle$ is finite. And by Theorem 2.10 no state repeats, hence the evaluation must eventually reach a stuck state σ . By Lemma 2.21, the stuck state must be a final state of the form $[c \mid v_2 \mid \square]$. □

Since the machines are reversible, each combinator can also be evaluated backwards.

Definition 2.23 (Π -backward evaluation). *The backward evaluation of $c : A \leftrightarrow B$ is $eval^\dagger(c) : B \rightarrow A$ where $eval^\dagger(c)(v_1) = v_2$ if $[c \mid v_1 \mid \square] \mapsto^{\dagger*} \langle c \mid v_2 \mid \square \rangle$*

Moreover, the forward and backward evaluation functions are inverses of each other, and hence both are reversible total functions.

Theorem 2.24 (Π -reversible). *\mathcal{U}_{Agda} For all c , v_1 , and v_2 , we have $eval(c)(v_1) = v_2$ iff $eval^\dagger(c)(v_2) = v_1$.*

¹The termination proof in Agda just relies on structural recursion on c .

Proof. For all c , if $eval(c)(v_1) = v_2$ then $\langle c \mid v_1 \mid \square \rangle \mapsto^* [c \mid v_2 \mid \square]$. Therefore $[c \mid v_2 \mid \square] \mapsto^{\dagger*} \langle c \mid v_1 \mid \square \rangle$, and $eval^{\dagger}(c)(v_2) = v_1$.

If $eval^{\dagger}(c)(v_2) = v_1$ then $[c \mid v_2 \mid \square] \mapsto^{\dagger*} \langle c \mid v_1 \mid \square \rangle$. Therefore $\langle c \mid v_1 \mid \square \rangle \mapsto^* [c \mid v_2 \mid \square]$, and $eval(c)(v_1) = v_2$.

□

2.2.3 BIG-STEP INTERPRETER OF Π

In this section, a more efficient and more readable specification of the evaluation function using a big-step interpreter is described. The details of the interpreter implementation are given in Fig. 2.6 where $interp(c)(v) \Downarrow v'$ means $(c \ v)$ evaluates to v' .

$$\frac{\text{c base combinator}}{interp(c)(v) \Downarrow \delta(c, v)} \quad \frac{interp(c_1)(v) \Downarrow w}{interp(c_1 \oplus c_2)(inj_1 \ v) \Downarrow inj_1 \ w} \quad \frac{interp(c_2)(v) \Downarrow w}{interp(c_1 \oplus c_2)(inj_2 \ v) \Downarrow inj_2 \ w}$$

$$\frac{}{interp(id \leftrightarrow)(v) \Downarrow v} \quad \frac{interp(c_1)(v) \Downarrow v_1 \quad interp(c_2)(v_1) \Downarrow v_2}{interp(c_1 \ ; \ c_2)(v) \Downarrow v_2} \quad \frac{interp(c_1)(v_1) \Downarrow w_1 \quad interp(c_2)(v_2) \Downarrow w_2}{interp(c_1 \otimes c_2)((v_1, v_2)) \Downarrow (w_1, w_2)}$$

Figure 2.6: Π -interpreter

For base combinators, the interpreter uses the δ function like in the machine semantics. For $c_1 \oplus c_2$, it calls the corresponding combinator depending on the given input. For $c_1 \otimes c_2$, it calls both combinators and collects the results into a pair. And for $c_1 \ ; \ c_2$, it calls c_1 and then feeds the result to c_2 . The interpreter can be implemented in a functional programming language more easily and more efficiently than the machine semantics defined in Section 2.2.2, and most importantly it computes the same results as the reversible abstract machine.

Theorem 2.25. \mathcal{U}_{Agda} For all $c : A \leftrightarrow B$ and $v : A$, $interp(c)(v) \Downarrow eval(c)(v)$.

Proof. By induction on c :

- c is a base combinator or $id \leftrightarrow$: for any $v : A$, $interp(c)(v) \Downarrow \delta(c, v)$ and $eval(c)(v) = \delta(c, v)$ ($\langle c \mid v \mid \square \rangle \mapsto_1 [c \mid \delta(c, v) \mid \square]$).
- $c = c_1 \oplus c_2$: It must be the case that $A = A_1 + A_2$ and $B = B_1 + B_2$. For any $v : A_1 + A_2$, there are two cases:

- $v = \mathbf{inj}_1 v_1$ for some $v_1 : A_1$: By Theorem 2.22 there exists $v'_1 : B_1$ such that $eval(c_1)(v_1) = v'_1$. Therefore

$$\begin{aligned} & \langle c_1 \oplus c_2 \mid \mathbf{inj}_1 v_1 \mid \square \rangle \\ \mapsto_4 & \langle c_1 \mid v_1 \mid (\square \oplus c_2) \bullet \square \rangle \mapsto^* [c_1 \mid v'_1 \mid (\square \oplus c_2) \bullet \square] \\ \mapsto_{11} & [c_1 \oplus c_2 \mid \mathbf{inj}_1 v'_1 \mid \square] \end{aligned}$$

Hence $eval(c_1 \oplus c_2)(\mathbf{inj}_1 v_1) = \mathbf{inj}_1 v'_1$. By the induction hypothesis $interp(c_1)(v_1) \Downarrow v'_1$, therefore

$$\frac{interp(c_1)(v_1) \Downarrow v'_1}{interp(c_1 \oplus c_2)(\mathbf{inj}_1 v_1) \Downarrow \mathbf{inj}_1 v'_1}$$

- $v_1 = \mathbf{inj}_2 v_2$ for some $v_2 : A_2$: Similar.

- $c = c_1 \otimes c_2$: It must be the case that $A = A_1 \times A_2$, $B = B_1 \times B_2$ and $v = (v_1, v_2)$. By Theorem 2.22 there exists v'_1 and v'_2 such that $eval(c_1)(v_1) = v'_1$ and $eval(c_2)(v_2) = v'_2$. Therefore

$$\begin{aligned} & \langle c_1 \otimes c_2 \mid (v_1, v_2) \mid \square \rangle \\ \mapsto_6 & \langle c_1 \mid v_1 \mid (\square \otimes [c_2, v_2]) \bullet \square \rangle \mapsto^* [c_1 \mid v'_1 \mid (\square \otimes [c_2, v_2]) \bullet \square] \\ \mapsto_8 & \langle c_2 \mid v_2 \mid ([c_1, v'_1] \otimes \square) \bullet \square \rangle \mapsto^* [c_2 \mid v'_2 \mid ([c_1, v'_1] \otimes \square) \bullet \square] \\ \mapsto_9 & [c_1 \otimes c_2 \mid (v'_1, v'_2) \mid \square] \end{aligned}$$

Hence $eval(c_1 \otimes c_2)(v_1, v_2) = (v'_1, v'_2)$. And by the induction hypothesis $interp(c_1)(v_1) \Downarrow v'_1$ and $interp(c_2)(v_2) \Downarrow v'_2$, therefore

$$\frac{interp(c_1)(v_1) \Downarrow v'_1 \quad interp(c_2)(v_2) \Downarrow v'_2}{interp(c_1 \otimes c_2)((v_1, v_2)) \Downarrow (v'_1, v'_2)}$$

- $c = c_1 \circledast c_2$: By Theorem 2.22 there exists v' and v'' such that $eval(c_1)(v) = v'$ and $eval(c_2)(v') = v''$. Therefore

$$\begin{aligned} & \langle c_1 \circledast c_2 \mid v \mid \square \rangle \\ \mapsto_3 & \langle c_1 \mid v \mid (\square \circledast c_2) \bullet \square \rangle \mapsto^* [c_1 \mid v' \mid (\square \circledast c_2) \bullet \square] \\ \mapsto_7 & \langle c_2 \mid v' \mid (c_1 \circledast \square) \bullet \kappa \rangle \mapsto^* \langle c_2 \mid v'' \mid (c_1 \circledast \square) \bullet \square \rangle \\ \mapsto_{10} & [c_1 \circledast c_2 \mid v'' \mid \square] \end{aligned}$$

Hence $eval(c_1 \circ c_2)(v) = v''$. And by the induction hypothesis $interp(c_1)(v) \Downarrow v'$ and $interp(c_2)(v') \Downarrow v''$, therefore

$$\frac{interp(c_1)(v) \Downarrow v' \quad interp(c_2)(v') \Downarrow v''}{interp(c_1 \circ c_2)(v) \Downarrow v''}$$

□

2.2.4 RIG GROUPOID

The syntax of Π comes from a rig groupoid. This section will show that the operational semantics given in Section 2.2.2 and 2.2.3 for Π forms a rig groupoid. Some definitions are needed before defining the category:

Definition 2.26. *Let \sim be the relation on combinators such that for all $c_1, c_2 : A \leftrightarrow B$, $c_1 \sim c_2$ iff $eval(c_1) = eval(c_2)$.*

It is obvious that \sim is an equivalence relation, which identifies combinators with the same behavior under evaluation.

Definition 2.27. *The notation $[c]_{\sim}$ refers to the \sim -equivalence class of c , where c is a representative combinator.*

Using the equivalence relation \sim , the category of Π can be defined: ²

Theorem 2.28. *$\mathcal{U}Agda \mathcal{C}^{\Pi}$ is a category where*

- *$Obj(\mathcal{C}^{\Pi})$ is the set of Π types,*
- *$Hom_{\mathcal{C}^{\Pi}}(A, B) = \{[c]_{\sim} \mid c : A \leftrightarrow B\}$*

Proof. The composition of \mathcal{C}^{Π} is defined as $[c_2]_{\sim} \circ [c_1]_{\sim} = [c_1 \circ c_2]_{\sim}$ for any $c_1 : A \leftrightarrow B$ and $c_2 : B \leftrightarrow C$. One important thing that needs to be checked is that composition respects the equivalence relation \sim , i.e. for all $c_1, c'_1 : A \leftrightarrow B$ and $c_2, c'_2 : B \leftrightarrow C$ if $c_1 \sim c'_1$ and $c_2 \sim c'_2$ then $c_1 \circ c_2 \sim c'_1 \circ c'_2$. This can be easily checked using the big-step interpreter semantics, for any $v_1 : A$ and $v_3 : C$, if $interp(c_1 \circ c_2, v_1) \Downarrow v_3$ then there exists $v_2 : B$ such that

$$\frac{interp(c_1)(v_1) \Downarrow v_2 \quad interp(c_2)(v_2) \Downarrow v_3}{interp(c_1 \circ c_2)(v_1) \Downarrow v_3}$$

²This approach is sufficient to prove the semantics forms a 1-category but ignores the rich structure at the next level [12].

Since $c_1 \sim c'_1$ and $c_2 \sim c'_2$ so $\text{interp}(c'_1)(v_1) \Downarrow v_2$ and $\text{interp}(c'_2)(v_2) \Downarrow v_3$, hence

$$\frac{\text{interp}(c'_1)(v_1) \Downarrow v_2 \quad \text{interp}(c'_2)(v_2) \Downarrow v_3}{\text{interp}(c'_1 \circledast c'_2)(v_1) \Downarrow v_3}$$

The other direction is similar.

After defining composition, there are two things left to check:

- Identity arrow: $[\text{id} \leftrightarrow]$ is the identity arrow for any Π type, it can be easily checked that for all $c : A \leftrightarrow B$, $[c \circledast \text{id} \leftrightarrow] = [c] = [\text{id} \leftrightarrow \circledast c]$ (i.e. $c \circledast \text{id} \leftrightarrow \sim c \sim \text{id} \leftrightarrow \circledast c$).
- Associativity: for any $c_1 : A \leftrightarrow B, c_2 : B \leftrightarrow C, c_3 : C \leftrightarrow D, v_1 : A$ and $v_4 : D$, if $\text{interp}((c_1 \circledast c_2) \circledast c_3, v_1) \Downarrow v_4$ then there exists $v_2 : B$ and $v_3 : C$ such that

$$\frac{\frac{\text{interp}(c_1)(v_1) \Downarrow v_2 \quad \text{interp}(c_2)(v_2) \Downarrow v_3}{\text{interp}(c_1 \circledast c_2)(v_1) \Downarrow v_2} \quad \text{interp}(c_3)(v_3) \Downarrow v_4}{\text{interp}((c_1 \circledast c_2) \circledast c_3)(v_1) \Downarrow v_4}$$

Hence

$$\frac{\text{interp}(c_1)(v_1) \Downarrow v_2 \quad \frac{\text{interp}(c_2)(v_2) \Downarrow v_3 \quad \text{interp}(c_3)(v_3) \Downarrow v_4}{\text{interp}(c_2 \circledast c_3)(v_3) \Downarrow v_4}}{\text{interp}(c_1 \circledast (c_2 \circledast c_3))(v_1) \Downarrow v_4}$$

The other direction is similar.

□

To show that \mathcal{C}^Π is a groupoid, the definition of inverse of combinators is needed:

$$\begin{aligned} !c &= c' && \text{if } c \text{ is base combinator, where } c' \text{ is } c\text{'s dual} \\ !\text{id} \leftrightarrow &= \text{id} \leftrightarrow \\ !(c_1 \oplus c_2) &= !c_1 \oplus !c_2 \\ !(c_1 \otimes c_2) &= !c_1 \otimes !c_2 \\ !(c_1 \circledast c_2) &= !c_2 \circledast !c_1 \end{aligned}$$

Theorem 2.29. *$\mathcal{U}\text{Agda } \mathcal{C}^\Pi$ is a groupoid.*

Proof. The inverse of an arrow can simply be defined as $[c]^{-1} = [!c]$. Two things need to be checked for the inverse:

- $c \circledast !c \sim \text{id} \leftrightarrow$: By induction on c :

- c is a base combinator: The dual of c computes its inverse, i.e. $\delta(!c, v) = \delta^\dagger(c, v)$.

Hence for all v :

$$\frac{\text{interp}(c)(v) \Downarrow \delta(c, v) \quad \text{interp}(!c)(\delta(c, v)) \Downarrow \delta^\dagger(c, \delta(c, v))}{\text{interp}(c \circledast !c)(v) \Downarrow v}$$

- $c = c_1 \oplus c_2$: For any v there are two cases:

- * $v = \text{inj}_1 x$: By induction, there exists x' such that

$$\frac{\text{interp}(c_1)(x) \Downarrow x' \quad \text{interp}(!c_1)(x') \Downarrow x}{\text{interp}(c_1 \circledast !c_1)(x) \Downarrow x}$$

Hence,

$$\frac{\frac{\text{interp}(c_1)(x) \Downarrow x'}{\text{interp}(c_1 \oplus c_2)(\text{inj}_1 x) \Downarrow \text{inj}_1 x'} \quad \frac{\text{interp}(!c_1)(x') \Downarrow x}{\text{interp}(!c_1 \oplus !c_2)(\text{inj}_1 x') \Downarrow \text{inj}_1 x}}{\text{interp}((c_1 \oplus c_2) \circledast (!c_1 \oplus !c_2))(\text{inj}_1 x) \Downarrow \text{inj}_1 x}}$$

- * $v = \text{inj}_2 y$: Similar.

- $c = c_1 \otimes c_2$: For any $v = (x, y)$, by induction there exist x' and y' such that

$$\frac{\text{interp}(c_1)(x) \Downarrow x' \quad \text{interp}(!c_1)(x') \Downarrow x}{\text{interp}(c_1 \circledast !c_1)(x) \Downarrow x} \quad \frac{\text{interp}(c_2)(y) \Downarrow y' \quad \text{interp}(!c_2)(y') \Downarrow y}{\text{interp}(c_2 \circledast !c_2)(y) \Downarrow y}$$

Hence,

$$\frac{\frac{\text{interp}(c_1)(x) \Downarrow x' \quad \text{interp}(c_2)(y) \Downarrow y'}{\text{interp}(c_1 \otimes c_2)((x, y)) \Downarrow (x', y')} \quad \frac{\text{interp}(!c_1)(x') \Downarrow x \quad \text{interp}(!c_2)(y') \Downarrow y}{\text{interp}(!c_1 \otimes !c_2)((x', y')) \Downarrow (x, y)}}{\text{interp}((c_1 \otimes c_2) \circledast (!c_1 \otimes !c_2))((x, y)) \Downarrow (x, y)}}$$

- $c = c_1 \circledast c_2$: Using the properties of the category from Theorem 2.28 and induction:

$$\begin{aligned} (c_1 \circledast c_2) \circledast (!c_2 \circledast !c_1) &\sim c_1 \circledast (c_2 \circledast (!c_2 \circledast !c_1)) \\ &\sim c_1 \circledast ((c_2 \circledast !c_2) \circledast !c_1) && \sim c_1 \circledast (\text{id} \leftrightarrow \circledast !c_1) && \text{by induction} \\ &\sim c_1 \circledast !c_1 && \sim \text{id} \leftrightarrow && \text{by induction} \end{aligned}$$

- $!c \circledast c \sim \text{id} \leftrightarrow$: Similar.

□

\mathcal{C}^Π contains two symmetric monoidal categories $(\mathcal{C}^\Pi, _ + _, 0)$ and $(\mathcal{C}^\Pi, _ \times _, \mathbf{1})$:

Theorem 2.30. $\mathcal{U}_{\text{Agda}}(\mathcal{C}^{\Pi}, _ + _, \mathbb{0})$ is a symmetric monoidal category.

Proof. The monoidal tensor takes two types A, B to $A + B$ and two arrows $[c_1], [c_2]$ to $[c_1 \oplus c_2]$. The associator is $[\text{assocl}_+] : t_1 + (t_2 + t_3) \leftrightarrow (t_1 + t_2) + t_3 : [\text{assocr}_+]$, the left unitor is $[\text{unite}_+l] : \mathbb{0} + t \leftrightarrow t : [\text{uniti}_+l]$, the right unitor is $[\text{unite}_+r] : t + \mathbb{0} \leftrightarrow t : [\text{uniti}_+r]$, and the braiding is $[\text{swap}_+]$. The coherence conditions are straightforward to check. \square

Theorem 2.31. $\mathcal{U}_{\text{Agda}}(\mathcal{C}^{\Pi}, _ \times _, \mathbb{1})$ is a symmetric monoidal category.

Proof. The monoidal tensor takes two types A, B to $A \times B$ and two arrows $[c_1], [c_2]$ to $[c_1 \otimes c_2]$. The associator is $[\text{assocl}_*] : t_1 \times (t_2 \times t_3) \leftrightarrow (t_1 \times t_2) \times t_3 : [\text{assocr}_*]$, the left unitor is $[\text{unite}_*l] : \mathbb{1} \times t \leftrightarrow t : [\text{uniti}_*l]$, the right unitor is $[\text{unite}_*r] : t \times \mathbb{1} \leftrightarrow t : [\text{uniti}_*r]$, and the braiding is $[\text{swap}_*]$. The coherence conditions are straightforward to check. \square

With distributivity and absorption isomorphisms, \mathcal{C}^{Π} is a rig groupoid.

Theorem 2.32. $\mathcal{U}_{\text{Agda}}(\mathcal{C}^{\Pi}, _ + _, \mathbb{0}, _ \times _, \mathbb{1})$ is a rig groupoid

Proof. The left distributivity isomorphism is $\text{distl} : t_1 \times (t_2 + t_3) \leftrightarrow (t_1 \times t_2) + (t_1 \times t_3) : \text{factorl}$, the right distributivity isomorphism is $\text{distr} : (t_1 + t_2) \times t_3 \leftrightarrow (t_1 \times t_3) + (t_2 \times t_3) : \text{factor}$, the left absorption isomorphism is $\text{absorbl} : \mathbb{0} \times t \leftrightarrow \mathbb{0} : \text{factorzr}$. and the right absorption isomorphism is $\text{absorbr} : t \times \mathbb{0} \leftrightarrow \mathbb{0} : \text{factorzl}$. The coherence conditions are straightforward to check. \square

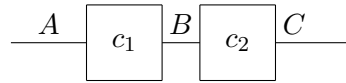
2.2.5 EXAMPLES

This section will introduce some example Π programs. To make Π programs easier to read a diagram representation will be introduced, and it will be used to represent Π programs throughout out the whole dissertation.

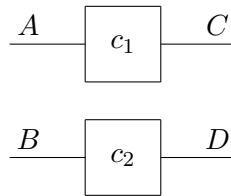
A wire will always be annotated with its type, product types will be represented as parallel wires and sum types will be represented as parallel wires with the symbol “+” between wires. Wires that have type $\mathbb{0}$ or $\mathbb{1}$ could be omitted in the diagram. The following will provide diagrams for all Π combinators. The most simple diagram is $\text{id} \leftrightarrow : A \leftrightarrow A$ which is just a wire:

A

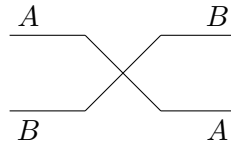
For $c_1 : A \leftrightarrow B$ and $c_2 : B \leftrightarrow C$, $c_1 ; c_2$ will be represented as two diagrams concatenated together:



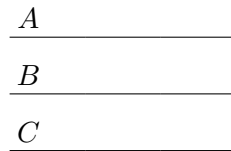
For $c_1 : A \leftrightarrow C$ and $c_2 : B \leftrightarrow D$, $c_1 \otimes c_2$ will be represented as two diagrams in parallel:



And $\text{swap}_* : A \times B \leftrightarrow B \times A$ will be represented as a cross:



The representations of assocl_* and assocr_* are identical to $\text{id} \leftrightarrow$ because the diagram representation does not contain the information about how the types are bind together:



And for unite_* and uniti_* it could be represented as either

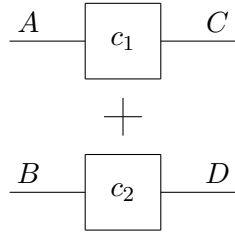


or

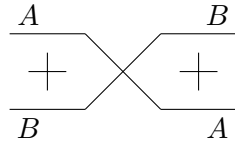


because the wire has type $\mathbb{1}$ could be omitted.

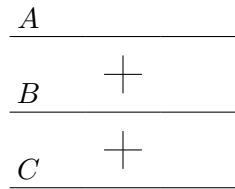
The combinators for sum types are similar. For $c_1 : A \leftrightarrow C$ and $c_2 : B \leftrightarrow D$, $c_1 \oplus c_2$ will be represented as two diagrams in parallel with the “+” symbol:



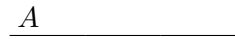
And $\text{swap}_+ : A + B \leftrightarrow B + A$ will be represented as a cross:



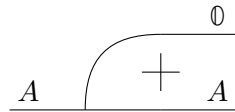
The representations of assocl_+ and assocr_+ are identical to $\text{id}\leftrightarrow$:



And for unite_+ and uniti_+ it could be represented as either



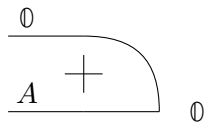
or



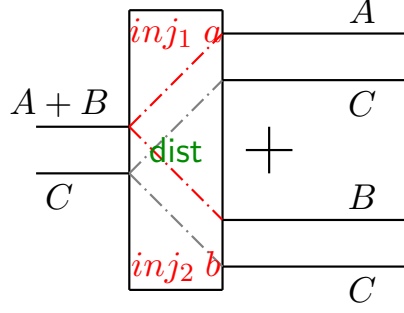
because the wire has type $\mathbb{0}$ could be omitted.

$\text{absorb}_r : \mathbb{0} \times A \leftrightarrow \mathbb{0}$: factor_l could be represented as either a blank diagram

or



The most complicated one is the diagram for $\text{dist} : (A + B) \times C$:



Note that in the diagram representation products have higher precedence than sums, so the right-hand side of the diagram represents type $(A + B) \times C$ not $A \times (C + B) \times C$.

The diagram representation introduced here is obviously not complete because it is incapable to represent $(A + B) \times C$ unless we squash $A + B$ into a single wire. However, it is expressive enough for describing all the Π programs in this dissertation. For a complete graphical language for bimonoidal categories, please refer to sheet diagrams proposed by Comfort et al. [18].

The following are some example Π programs, which will become important building blocks in the construction of more complicated programs in the following chapters. First, to make things easier when building larger circuits, some adapters which deal with routing are needed:

$$[A+B]+C=[C+B]+A : \forall \{A B C\} \rightarrow (A +_u B) +_u C \leftrightarrow (C +_u B) +_u A$$

$$[A+B]+C=[C+B]+A = \text{assocr}_+ \ ; \ (\text{id} \leftrightarrow \oplus \ \text{swap}_+) \ ; \ \text{swap}_+$$

$$[A+B]+C=[A+C]+B : \forall \{A B C\} \rightarrow (A +_u B) +_u C \leftrightarrow (A +_u C) +_u B$$

$$[A+B]+C=[A+C]+B = \text{assocr}_+ \ ; \ (\text{id} \leftrightarrow \oplus \ \text{swap}_+) \ ; \ \text{assocl}_+$$

$$[A+B]+[C+D]=[A+C]+[B+D] : \{A B C D : \mathbb{U}\} \rightarrow (A +_u B) +_u (C +_u D) \leftrightarrow (A +_u C) +_u (B +_u D)$$

$$[A+B]+[C+D]=[A+C]+[B+D] = \text{assocl}_+ \ ; \ (\text{assocr}_+ \oplus \ \text{id} \leftrightarrow) \ ; \ ((\text{id} \leftrightarrow \oplus \ \text{swap}_+) \oplus \ \text{id} \leftrightarrow) \ ; \ (\text{assocl}_+ \oplus \ \text{id} \leftrightarrow) \ ; \ \text{assocr}_+$$

$$A+[B+C]=B+[A+C] : \forall \{A B C\} \rightarrow A +_u (B +_u C) \leftrightarrow B +_u (A +_u C)$$

$$A+[B+C]=B+[A+C] = \text{assocl}_+ \ ; \ \text{swap}_+ \oplus \ \text{id} \leftrightarrow \ ; \ \text{assocr}_+$$

$$A \times [B \times C] = B \times [A \times C] : \{A B C : \mathbb{U}\} \rightarrow A \times_u (B \times_u C) \leftrightarrow B \times_u (A \times_u C)$$

$$A \times [B \times C] = B \times [A \times C] = \text{assocl}_* \ ; \ (\text{swap}_* \otimes \ \text{id} \leftrightarrow) \ ; \ \text{assocr}_*$$

$$[A \times B] \times C = [A \times C] \times B : \forall \{A B C\} \rightarrow (A \times_u B) \times_u C \leftrightarrow (A \times_u C) \times_u B$$

$$[A \times B] \times C = [A \times C] \times B = \text{assocr}_* \ ; \ (\text{id} \leftrightarrow \otimes \ \text{swap}_*) \ ; \ \text{assocl}_*$$

$$[A \times B] \times [C \times D] = [A \times C] \times [B \times D] : \{A B C D : \mathbb{U}\} \rightarrow (A \times_u B) \times_u (C \times_u D) \leftrightarrow (A \times_u C) \times_u (B \times_u D)$$

$$[A \times B] \times [C \times D] = [A \times C] \times [B \times D] = \text{assocl}_* \ ; \ (\text{assocr}_* \otimes \ \text{id} \leftrightarrow) \ ; \ ((\text{id} \leftrightarrow \otimes \ \text{swap}_*) \otimes \ \text{id} \leftrightarrow) \ ; \ (\text{assocl}_* \otimes \ \text{id} \leftrightarrow) \ ; \ \text{assocr}_*$$

These adapters will be very useful in the more complicated circuits in Chapters 3 and 5.

The next thing is to define bits, which will be used in most of the examples in the whole dissertation. As described in Section 2.2.1 a boolean value will be encoded using $\mathbb{1} + \mathbb{1}$:

$$\mathbb{B} : \mathbb{U}$$

$$\mathbb{B} = \mathbb{1} +_u \mathbb{1}$$

$$\text{pattern } \mathbb{F} = \text{inj}_1 \text{ tt}$$

$$\text{pattern } \mathbb{T} = \text{inj}_2 \text{ tt}$$

where \mathbb{U} is the type universe of Π . More generally, a collection of n -bits can be represented as a product of n - \mathbb{B} values:

$$\mathbb{B}^\wedge : \mathbb{N} \rightarrow \mathbb{U}$$

$$\mathbb{B}^\wedge 0 = \mathbb{1}$$

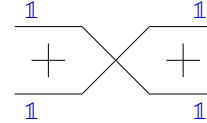
$$\mathbb{B}^\wedge 1 = \mathbb{B}$$

$$\mathbb{B}^\wedge (\text{succ } (\text{succ } n)) = \mathbb{B} \times_u \mathbb{B}^\wedge (\text{succ } n)$$

The simplest operation on \mathbb{B} is NOT, which flips the value:

$$\text{NOT} : \mathbb{B} \leftrightarrow \mathbb{B}$$

$$\text{NOT} = \text{swap}_+$$



The most common use of \mathbb{B} is performing conditional operations, which can be achieved using **dist** and **factor**:

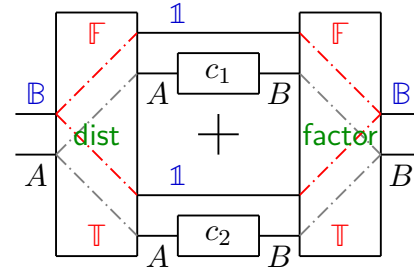
$$\text{IF} : \forall \{A B\} \rightarrow (c_1 c_2 : A \leftrightarrow B)$$

$$\rightarrow \mathbb{B} \times_u A \leftrightarrow \mathbb{B} \times_u B$$

$$\text{IF } c_1 c_2 = \text{dist} \ ;$$

$$((\text{id} \leftrightarrow \otimes c_1) \oplus (\text{id} \leftrightarrow \otimes c_2)) \ ;$$

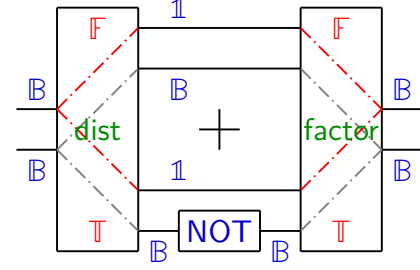
$$\text{factor}$$



If the input is $(\mathbb{F}, a)/(\mathbb{T}, a)$, after **dist** it will become $\text{inj}_1 (\text{tt}, a)/\text{inj}_2 (\text{tt}, a)$, and c_1/c_2 will be executed to obtain $\text{inj}_1 (\text{tt}, b)/\text{inj}_2 (\text{tt}, b)$, and **factor** will restore \mathbb{F}/\mathbb{T} from the $\text{inj}_1 / \text{inj}_2$ which results in $(\mathbb{F}, b)/(\mathbb{T}, b)$.

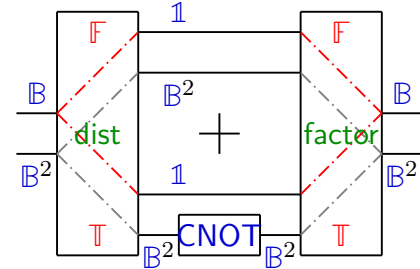
Using **IF**, **CNOT** which is an important gate in reversible and quantum computation can be easily implemented. It will execute $\text{id} \leftrightarrow \text{NOT}$ if the control bit is **F/T**:

$$\begin{aligned} \text{CNOT} &: \mathbb{B}^2 \leftrightarrow \mathbb{B}^2 \\ \text{CNOT} &= \text{IF } \text{id} \leftrightarrow \text{NOT} \end{aligned}$$



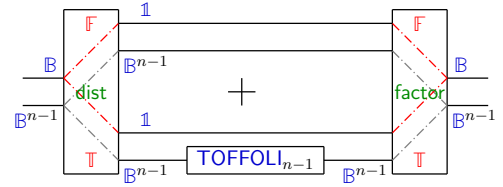
Iterating this again, **toffoli** can be obtained:

$$\begin{aligned} \text{toffoli} &: \mathbb{B}^3 \leftrightarrow \mathbb{B}^3 \\ \text{toffoli} &= \text{IF } \text{id} \leftrightarrow \text{CNOT} \end{aligned}$$



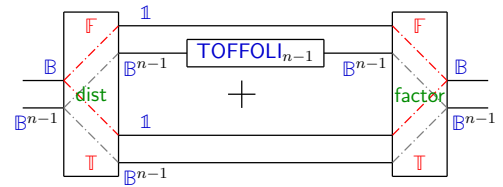
Hence, the n -bit Toffoli ($\text{TOFFOLI}_n(b_1, \dots, b_{n-1}, b) = (b_1, \dots, b_{n-1}, b \text{ xor } \bigwedge_{i=1}^n b_i)$) can be constructed recursively:

$$\begin{aligned} \text{TOFFOLI} &: \forall \{n\} \rightarrow \mathbb{B}^n \leftrightarrow \mathbb{B}^n \\ \text{TOFFOLI} \{0\} &= \text{id} \leftrightarrow \\ \text{TOFFOLI} \{1\} &= \text{NOT} \\ \text{TOFFOLI} \{\text{suc}(\text{suc } n)\} &= \text{IF } \text{id} \leftrightarrow \text{TOFFOLI} \end{aligned}$$



And with a little tweak, $\text{TOFFOLI}'_n(b_1, \dots, b_{n-1}, b) = (b_1, \dots, b_{n-1}, b \text{ xor } \bigwedge_{i=1}^n \bar{b}_i)$ can also be constructed:

$$\begin{aligned} \text{TOFFOLI}' &: \forall \{n\} \rightarrow \mathbb{B}^n \leftrightarrow \mathbb{B}^n \\ \text{TOFFOLI}' \{0\} &= \text{id} \leftrightarrow \\ \text{TOFFOLI}' \{1\} &= \text{NOT} \\ \text{TOFFOLI}' \{\text{suc}(\text{suc } n)\} &= \text{IF } \text{TOFFOLI}' \text{id} \leftrightarrow \end{aligned}$$



The final example of this section is a binary incremter **INCR**, which will increment an n -bit binary number (without carry). The main idea is using INCR_{n-1} to increment the lower $n - 1$ -bits. If after incrementing, the lower $n - 1$ -bits are all **F** then the highest bit is flipped. To implement this, first a function for moving bits around is needed, **FST2LAST** will move the first bit to the last i.e. $\text{FST2LAST}_n(b_1, b_2, \dots, b_n) = (b_2, \dots, b_n, b_1)$:

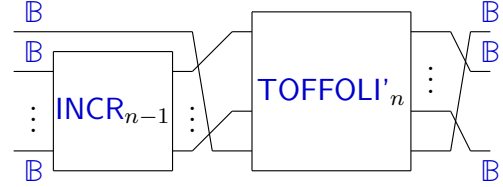
identifier	$id ::= string$
type	$t ::= id \mid (t, \dots)$
data type declaration	$dcl ::= \mathbf{data} \ id \ [= \ constrs]$
constructors	$constrs ::= id \ [t] \ [\mid \ constrs]$
function declaration	$f ::= id \ :: \ t \ \leftrightarrow \ t \ \backslash n \ [body]$
function body	$body ::= id \ pat \ = \ pat \ \backslash n \ [body]$
pattern	$pat ::= id \ \mid \ (pat, \dots) \ \mid \ id \ [pat]$

Figure 2.7: Syntax of Π^{++}

$FST2LAST : \forall \{n\} \rightarrow \mathbb{B}^{\wedge} n \leftrightarrow \mathbb{B}^{\wedge} n$
 $FST2LAST \{0\} = id \leftrightarrow$
 $FST2LAST \{1\} = id \leftrightarrow$
 $FST2LAST \{2\} = swap_*$
 $FST2LAST \{suc \ (suc \ (suc \ n))\} = Ax[BxC]=Bx[AxC] \ ; \ (id \leftrightarrow \otimes FST2LAST)$

Using $FST2LAST$ and $TOFFOLI'$, $INCR$ can be implemented easily:

$INCR : \forall \{n\} \rightarrow \mathbb{B}^{\wedge} n \leftrightarrow \mathbb{B}^{\wedge} n$
 $INCR \{0\} = id \leftrightarrow$
 $INCR \{1\} = swap_+$
 $INCR \{suc \ (suc \ n)\} =$
 $(id \leftrightarrow \otimes INCR) \ ; \ FST2LAST \ ; \ TOFFOLI' \ ; \ FST2LAST^{-1}$



2.3 INTRODUCTION TO Π^{++}

Π^{++} is a reversible programming language, which provides Haskell-like syntax over Π . The main features of Π^{++} are algebraic data types and pattern matching, which makes programming easier. The syntax of Π^{++} is given in Fig. 2.7. The types of Π^{++} are either sum (data types) or product types:

- Data type declarations: The syntax is similar to Haskell, each data type declaration contains an arbitrary number of constructors. And each constructor has zero or one type argument. This might seem like a restriction, but since the language includes product types, constructors can take more type arguments by packing them into a product type. The following are some example data type declarations:

```

data Zero
data One = TT
data B = F | T
data B2 = B2 ( B , B )

```

Constructors of data types can be used in pattern matching clauses. For example, this is the **NOT** function implemented in Π^{++} :

```

not :: B <-> B
not T = F
not F = T

```

- Product types: Product types can be represented as tuples, which do not need declaration, which can be pattern matched using tuple pattern. For example, this is the **CNOT** function implemented in Π^{++} :

```

cnot :: ( B , B ) <-> ( B , B )
cnot ( F , b ) = ( F , b )
cnot ( T , b ) = ( T , not b )

```

Π^{++} will check for overlapping patterns on both left and right sides of the function declarations, and signals errors. For example, the following function contains overlapping patterns:

```

err :: ( B , B ) <-> ( B , B )
err ( b , F ) = ( F , b )
err ( T , b ) = ( T , not b )

```

where (b , F) and (T , b) overlap since both patterns contain the value (T , F) . The language will also check for non-exhaustive patterns to guarantee reversibility. For example, the following function has non-exhaustive patterns:

```

err1 :: ( B , B ) <-> ( B , B )
err1 ( F , b ) = ( F , T )
err1 ( T , b ) = ( T , not b )

```

where the value (F , F) is not matched by patterns on the right-hand side.

Since Π is complete for representing all permutations on finite types [21,23], it is possible to compile Π^{++} into Π . However, how to compile Π^{++} to Π efficiently is still an open problem. Even though programming in Π^{++} is easier than Π , the formalization of Π in Agda provides a powerful meta-programming mechanism of Π . Hence in this dissertation, Π^{++} and Π will be used interchangeably.

2.4 SPACE-TIME TRADE-OFF

In the next two chapters, two extensions of Π will be presented, each yielding a distinct compact closed category: one of these categories will have a dual $-$ for the type constructor $+$ and the other a dual $/$ for the type constructor \times . This section develops the intuition that the type constructor $+$ is related to the time needed for runtime execution and that the type constructor \times is related to the space needed for runtime execution thus setting the stage for an interpretation of $-$ as “going backward in time” (backtracking of control flow) and of $/$ as “going backward in space” (reclaiming of allocated storage).

Given the small-step abstract machine introduced in Section 2.2.2, it is relatively straightforward to compute the time and space resources needed by a computation: “time” is modeled by the number of machine transition steps, and an upper bound on “space” resources is modeled by the maximum size of intermediate machine states visited during execution. A simple abstract measure of the size of a machine state $\#\sigma$ is the number of values stored in that state (whether in the v register or the continuation register κ). This measure essentially counts the number of live “pebbles” in the “pebble game” models of reversible computation [7,13] and is defined as follows:

$$\begin{array}{ll}
\#\sigma = 1 + \#\sigma.\kappa & \#\square = 0 \\
\#(\square \mathbin{;} c_2) \bullet \kappa = \#\kappa & \#(c_1 \mathbin{;} \square) \bullet \kappa = \#\kappa \\
\#(\square \oplus c_2) \bullet \kappa = \#\kappa & \#(c_1 \oplus \square) \bullet \kappa = \#\kappa \\
\#(\square \otimes [c_2, v]) \bullet \kappa = 1 + \#\kappa & \#[[c_1, v] \otimes \square] \bullet \kappa = 1 + \#\kappa
\end{array}$$

Since every state has a value register, its size is 1 plus the size of the continuation register. This measure ignores the (fixed and constant) space needed to store the program itself (i.e., the combinators occurring in either the c component or the κ component) and focuses on the maximum dynamic space requirements needed for each state during execution.

To make the case that the number of machine transition steps (i.e., time) is related to the $+$ type constructor and that the space used by intermediate machine states is related to the \times type constructor, let's look at a small example. A type containing 16 values can be represented as a sum of 8 booleans $\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + \mathbb{B}))))))$ or the product of 4 booleans $\mathbb{B} \times (\mathbb{B} \times (\mathbb{B} \times \mathbb{B}))$. In both cases, we can distinguish one boolean b and consider it indexed by the remaining 8 values as shown below:

$(\text{inj}_1 b)$	$(\mathbb{F}, (\mathbb{F}, (\mathbb{F}, b)))$
$(\text{inj}_2 (\text{inj}_1 b))$	$(\mathbb{F}, (\mathbb{F}, (\mathbb{T}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_1 b)))$	$(\mathbb{F}, (\mathbb{T}, (\mathbb{F}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_1 b))))$	$(\mathbb{F}, (\mathbb{T}, (\mathbb{T}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_1 b)))))$	$(\mathbb{T}, (\mathbb{F}, (\mathbb{F}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_1 b))))))$	$(\mathbb{T}, (\mathbb{F}, (\mathbb{T}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_1 b))))))$	$(\mathbb{T}, (\mathbb{T}, (\mathbb{F}, b)))$
$(\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 (\text{inj}_2 b))))))$	$(\mathbb{T}, (\mathbb{T}, (\mathbb{T}, b)))$

On the left, the boolean b is indexed by its position in the wide sum type; on the right, the boolean b is indexed by the values of the three other booleans. The two representations are in 1-1 correspondence and hence it is possible to write a Π -combinator that mediates between the two representations:

```
--  $\mathbb{B} + n = \mathbb{B} +_u \dots +_u \mathbb{B}$ 
 $\mathbb{B} + : \mathbb{N} \rightarrow \mathbb{U}$ 
 $\mathbb{B} + 0 = \mathbb{0}$ 
 $\mathbb{B} + 1 = \mathbb{B}$ 
 $\mathbb{B} + (\text{suc } (\text{suc } n)) = \mathbb{B} +_u (\mathbb{B} + (\text{suc } n))$ 

convert :  $\forall \{n\} \rightarrow \mathbb{B} + (2 \wedge n) \leftrightarrow \mathbb{B} \wedge (1 + n)$ 
convert {0} = id $\leftrightarrow$ 
convert {1} = (uniti $_{*!} \oplus$  uniti $_{*!}$ ) ; factor
convert {suc (suc n)} = split ;
                                (convert {suc n}  $\oplus$  (coe {n} ; convert {suc n})) ;
                                (uniti $_{*!} \oplus$  uniti $_{*!}$ ) ; factor
```

where

$$\begin{aligned} \text{coe} &: \forall \{n\} \rightarrow \mathbb{B}+ ((2 \wedge n) + ((2 \wedge n) + 0) + 0) \leftrightarrow \mathbb{B}+ (2 \wedge (1 + n)) \\ \text{split} &: \forall \{n\} \{m\} \rightarrow \mathbb{B}+ (n + m) \leftrightarrow (\mathbb{B}+ n +_u \mathbb{B}+ m) \end{aligned}$$

Now consider two programs that negate the distinguished b and apply them to the inputs in the last line:

$$\begin{aligned} \text{flip+} &: (n : \mathbb{N}) \rightarrow \mathbb{B}+ n \leftrightarrow \mathbb{B}+ n \\ \text{flip+ } 0 &= \text{id}\leftrightarrow \\ \text{flip+ } 1 &= \text{swap}_+ \\ \text{flip+ } (\text{suc } (\text{suc } n)) &= \text{id}\leftrightarrow \oplus \text{flip+ } (\text{suc } n) \\ \\ \text{flip*} &: (n : \mathbb{N}) \rightarrow (\mathbb{B} \wedge n) \leftrightarrow (\mathbb{B} \wedge n) \\ \text{flip* } 0 &= \text{id}\leftrightarrow \\ \text{flip* } 1 &= \text{swap}_+ \\ \text{flip* } (\text{suc } (\text{suc } n)) &= \text{dist } \wp (\text{id}\leftrightarrow \oplus (\text{id}\leftrightarrow \otimes \text{flip* } (\text{suc } n))) \wp \text{factor} \end{aligned}$$

The big difference between them is in the amount of resources they use. Accessing b in the additive representation on the left requires “time” to decode the sum type; accessing b in the multiplicative representation on the right is immediate but requires additional space to store the three booleans. Indeed, when indexed by integers from 0 to 511, the additive program takes 1024 steps using 1 unit of space, whereas the multiplicative program takes 128 steps using 10 units of space. Generally, the additive approach takes $O(n)$ time using just one unit of space whereas the multiplicative approach takes $O(\log n)$ time at the expense of using $O(\log n)$ units of space.

CHAPTER 3

NEGATIVE TYPES

Aiming to construct a compact closed category with an additive dual, this chapter will propose a reversible programming language Π^- which is an extension of Π with two combinators η_+ and ε_+ witnessing the isomorphism $\mathbb{0} \leftrightarrow A + (-A)$. As explained in Section 2.2 the value of a sum type $A + B$ represents choices that are exercised at different times: at one time it might be a value of type A and at another time it might be a value of type B . In the case of A and $-A$, the two options cancel each other in some kind of destructive interference. This suggests that values should be equipped with a wave-like (but degenerate) notion of amplitude or phase. Therefore in this extension, a value v will be equipped with either a positive sign (which is omitted by convention) or a negative sign $-v$. Semantically the negative sign is interpreted as flipping the flow of evaluation: forward evaluation becomes backward evaluation and vice-versa. And as the coherence conditions of compact closed categories suggest, flipping the flow of evaluation twice cancels each other out. Operationally, since Π -abstract machines are reversible, all that is needed is using the backward evaluation relation \mapsto^\dagger for negative values.

The structure of this chapter is given as follows:

- Section 3.1 will introduce the syntax of Π^- .
- Section 3.2 will introduce the abstract machine semantics of Π^- , and study its properties.
- Section 3.3 will describe a big-step interpreter of Π^- which provides an efficient implementation.
- Section 3.4 will show that Π^- does form a compact closed category.

- Section 3.5 will provide some example Π^- programs, which demonstrate the usage of negative types.

3.1 SYNTAX OF Π^-

The syntax of Π^- extends Π via adding negative types and combinators η_+ and ε_+ , and is given in Fig. 3.1.

<i>Value types</i>	$t ::= 0 \mid \mathbb{1} \mid t + t \mid t \times t \mid -t$
<i>Values</i>	$v ::= \mathbf{tt} \mid \mathbf{inj}_1 v \mid \mathbf{inj}_2 v \mid (v, v) \mid -v$
<i>Combinator types</i>	$t \leftrightarrow t$
<i>Combinators</i>	$c ::= \mathbf{unite}_+ \mid \mathbf{unit}_+ \mid \mathbf{swap}_+ \mid \mathbf{assocl}_+ \mid \mathbf{assocr}_+$ $\mid \mathbf{unite}_* \mid \mathbf{unit}_* \mid \mathbf{swap}_* \mid \mathbf{assocl}_* \mid \mathbf{assocr}_* \mid \mathbf{absorbr} \mid \mathbf{factorzl}$ $\mid \mathbf{dist} \mid \mathbf{factor} \mid \mathbf{id} \leftrightarrow \mid c \circ c \mid c \oplus c \mid c \otimes c \mid \eta_+ \mid \varepsilon_+$
<i>Programs</i>	$p ::= c v$

Figure 3.1: Π^- syntax.

The inhabitants of a negative type $-t$ are just the inhabitants of type t labeled with a negative sign. The typing judgments of values are given in Fig. 3.2. And the typing judgments of combinators are collected in Fig. 3.3, where η_+ and ε_+ have type $0 \leftrightarrow A + (-A)$. With the two new combinators, Π^- contains the complete syntax from compact closed categories. In Sec. 3.2 the operational semantics of Π^- will be given, and Sec. 3.4 will show that it indeed forms a compact closed category.

$\mathbf{tt} : \mathbb{1}$	$\frac{v_1 : t_1 \quad v_2 : t_2}{(v_1, v_2) : t_1 \times t_2}$	$\frac{v_1 : t_1}{\mathbf{inj}_1 v_1 : t_1 + t_2}$	$\frac{v_2 : t_2}{\mathbf{inj}_2 v_2 : t_1 + t_2}$	$\frac{v : t}{-v : -t}$
----------------------------	---	--	--	-------------------------

Figure 3.2: Typing judgments of Π^- values.

3.2 ABSTRACT MACHINE SEMANTICS OF Π^-

This section will give an abstract machine semantics for Π^- . All combinators from Π will still have the same operational semantics as Π , what's left are the two new combinators η_+ and ε_+ . One way to figure out how η_+ and ε_+ are considering their types:

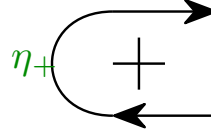
- It is impossible to evaluate η_+ in the forward direction as this would require supplying

$\text{id}\leftrightarrow :$	$t \leftrightarrow t$	$: \text{id}\leftrightarrow$
$\text{unite}_+ \text{l} :$	$\mathbb{0} + t \leftrightarrow t$	$: \text{uniti}_+ \text{l}$
$\text{swap}_+ :$	$t_1 + t_2 \leftrightarrow t_2 + t_1$	$: \text{swap}_+$
$\text{assocl}_+ :$	$t_1 + (t_2 + t_3) \leftrightarrow (t_1 + t_2) + t_3$	$: \text{assocr}_+$
$\text{unite}_* \text{l} :$	$\mathbb{1} \times t \leftrightarrow t$	$: \text{uniti}_* \text{l}$
$\text{swap}_* :$	$t_1 \times t_2 \leftrightarrow t_2 \times t_1$	$: \text{swap}_*$
$\text{assocl}_* :$	$t_1 \times (t_2 \times t_3) \leftrightarrow (t_1 \times t_2) \times t_3$	$: \text{assocr}_*$
$\text{absorbr} :$	$\mathbb{0} \times t \leftrightarrow \mathbb{0}$	$: \text{factorz}$
$\text{dist} :$	$(t_1 + t_2) \times t_3 \leftrightarrow (t_1 \times t_3) + (t_2 \times t_3)$	$: \text{factor}$
$\eta_+ :$	$\mathbb{0} \leftrightarrow t + (-t)$	$: \varepsilon_+$
$\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_2 \leftrightarrow t_3}{\vdash c_1 \circ c_2 : t_1 \leftrightarrow t_3} \quad \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \oplus c_2 : t_1 + t_3 \leftrightarrow t_2 + t_4} \quad \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \otimes c_2 : t_1 \times t_3 \leftrightarrow t_2 \times t_4}$		

Figure 3.3: Type judgments of Π^- combinators.

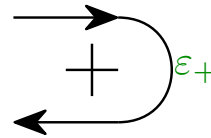
a value of the empty type $\mathbb{0}$. Hence it can only evaluate in the backward direction.

Graphically, the combinator η_+ looks like “U-turns”:



As the graph suggests, in the backward direction η_+ either expects a value $\text{inj}_1 v$ or a value $\text{inj}_2 (-v)$. In either case, it needs to flip the flow of evaluation with the appropriate value because there is no way to generate an inhabitant of $\mathbb{0}$.

- Similarly, it is impossible to evaluate ε_+ in the backward direction. Graphically, the combinator ε_+ also looks like a “U-turn” but in a different direction:



As the graph suggests, in the forward direction ε_+ either expects a value $\text{inj}_1 v$ or a value $\text{inj}_2 (-v)$. In either case, it needs to flip the flow of evaluation with the appropriate value.

To manage the additional expressiveness of negative types, we extend machine states to internally maintain a direction \triangleright or \triangleleft . The definition of extended machine states follows.

Definition 3.1 (Π^- -machine states). A Π^- -machine state σ is either:

- An enter state: $\langle c \mid v \mid \kappa \rangle_d$ where $c : A \leftrightarrow B$, $v : A$, $\kappa : \text{CONT}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.
- A return state: $[c \mid v \mid \kappa]_d$ where $c : A \leftrightarrow B$, $v : B$, $\kappa : \text{CONT}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.

Fig. 3.4 gives the transition rules of the abstract machine. It contains all of the transition rules from Π for states in the forward direction and inverse transition rules for states in the backward direction. The remaining four transition rules deal with the new combinators η_+ and ε_+ . As there are no values of type $\mathbb{0}$, it is impossible for an enter state to make progress on η_+ and it is impossible for a return state to make progress on ε_+ . For ε_+ it is only possible to make progress when entering the state in the forward direction resulting in a state that executes in the backward direction, swapping the direction of the value and the injection tag in the sum type. Symmetrically, a return state with η_+ in the combinator position does not return anything to the continuation but instead flips the direction of execution again. Final states include the usual final states from the Π -abstract machine but also “backward states” of the form $\langle c \mid v \mid \square \rangle_{\triangleleft}$. Such final states can be proper results for programs with negative types. For example, evaluating $\langle \varepsilon_+ \mid \text{inj}_1 \text{ tt} \mid \square \rangle_{\triangleright}$ proceeds using \mapsto_{13} to $\langle \varepsilon_+ \mid \text{inj}_2 (-\text{tt}) \mid \square \rangle_{\triangleleft}$ from which no further transitions are possible.

For base combinators c :

$$\begin{array}{l}
\langle c \mid v \mid \kappa \rangle_{\triangleright} \mapsto_1 [c \mid \delta(c, v) \mid \kappa]_{\triangleright} \qquad [c \mid v \mid \kappa]_{\triangleleft} \mapsto_1 \langle c \mid \delta^\dagger(c, v) \mid \kappa \rangle_{\triangleleft} \\
\langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle_{\triangleright} \mapsto_2 [\text{id} \leftrightarrow \mid v \mid \kappa]_{\triangleright} \qquad [\text{id} \leftrightarrow \mid v \mid \kappa]_{\triangleleft} \mapsto_2 \langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle_{\triangleleft} \\
\langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleright} \mapsto_3 \langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleleft} \mapsto_3 \langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleleft} \\
\langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleright} \mapsto_4 \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleleft} \mapsto_4 \langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleleft} \\
\langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleright} \mapsto_5 \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleleft} \mapsto_5 \langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleleft} \\
\langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleright} \mapsto_6 \langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleleft} \mapsto_6 \langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleleft} \\
\langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleright} \mapsto_7 \langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleleft} \mapsto_7 \langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleleft} \\
\langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleright} \mapsto_8 \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleright} \qquad \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleleft} \mapsto_8 \langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleleft} \\
\langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleright} \mapsto_9 \langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleright} \qquad \langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleleft} \mapsto_9 \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleleft} \\
\langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleright} \mapsto_{10} \langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleright} \qquad \langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleleft} \mapsto_{10} \langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleleft} \\
\langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleright} \mapsto_{11} \langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleright} \qquad \langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleleft} \mapsto_{11} \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleleft} \\
\langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleright} \mapsto_{12} \langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleright} \qquad \langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleleft} \mapsto_{12} \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleleft} \\
\langle \varepsilon_+ \mid \text{inj}_1 v \mid \kappa \rangle_{\triangleright} \mapsto_{13} \langle \varepsilon_+ \mid \text{inj}_2 (-v) \mid \kappa \rangle_{\triangleleft} \qquad [\eta_+ \mid \text{inj}_2 (-v) \mid \kappa]_{\triangleleft} \mapsto_{15} [\eta_+ \mid \text{inj}_1 v \mid \kappa]_{\triangleright} \\
\langle \varepsilon_+ \mid \text{inj}_2 (-v) \mid \kappa \rangle_{\triangleright} \mapsto_{14} \langle \varepsilon_+ \mid \text{inj}_1 v \mid \kappa \rangle_{\triangleleft} \qquad [\eta_+ \mid \text{inj}_1 v \mid \kappa]_{\triangleleft} \mapsto_{16} [\eta_+ \mid \text{inj}_2 (-v) \mid \kappa]_{\triangleright}
\end{array}$$

Figure 3.4: Π^- -abstract machine transition rules

The machine transition relation is both forward and backward deterministic.

Lemma 3.2 (Π^- -forward deterministic). $\mathcal{U}^{\text{Agda}}$ If $\sigma \mapsto \sigma_1$ and $\sigma \mapsto \sigma_2$ then $\sigma_1 = \sigma_2$

Proof. By checking all transitions. □

Lemma 3.3 (Π^- -backward deterministic). $\mathcal{U}Agda$ *If $\sigma_1 \mapsto \sigma$ and $\sigma_2 \mapsto \sigma$ then $\sigma_1 = \sigma_2$*

Proof. By checking all transitions. \square

Hence the machine is a reversible abstract machine. However, there are two possibilities for stuck states now:

Lemma 3.4 (Π^- -stuck states). $\mathcal{U}Agda$ *If σ is stuck, then either $\sigma = \langle c \mid v \mid \square \rangle_{\triangleleft}$ or $\sigma = [c \mid v \mid \square]_{\triangleright}$.*

Proof. By case analysis on all possible states. \square

So the definition of evaluation for Π^- needs to consider the two possible outcomes. To incorporate this, the composite set $(A^\rightarrow, B^\leftarrow)$ is needed, whose elements can be either a value $v : A^\rightarrow$ flowing in the forward direction or a value $w : B^\leftarrow$ flowing in the backward direction. Similar to stuck states, the initial states of Π^- also have two possible forms ($\langle c \mid v \mid \square \rangle_{\triangleright}$ and $[c \mid v \mid \square]_{\triangleleft}$) now. Hence the definition of evaluation will also need to take care of this, which is given as follows:

Definition 3.5 (Π^- -forward evaluation). *The evaluation of $c : A \leftrightarrow B$ is*

$$eval^-(c) : (A^\rightarrow, B^\leftarrow) \rightarrow (B^\rightarrow, A^\leftarrow)$$

where

$$eval^-(c)(v : A^\rightarrow) = \begin{cases} w : B^\rightarrow & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c \mid w \mid \square]_{\triangleright} \\ w : A^\leftarrow & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* \langle c \mid w \mid \square \rangle_{\triangleleft} \end{cases}$$

$$eval^-(c)(v : B^\leftarrow) = \begin{cases} w : B^\rightarrow & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^* [c \mid w \mid \square]_{\triangleright} \\ w : A^\leftarrow & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^* \langle c \mid w \mid \square \rangle_{\triangleleft} \end{cases}$$

Using Lemma 3.4 and the non-repeating theorem for reversible abstract machine, the totality of $eval^-(c)$ can be proved:

Theorem 3.6 (Π^- -termination). *For all Π^- -combinators $c : A \leftrightarrow B$ and $v_1 : (A^\rightarrow, B^\leftarrow)$ there exists $v_2 : (B^\rightarrow, A^\leftarrow)$ such that $eval(c)(v_1) = v_2$*

Proof. The set of reachable states starting from $\langle c \mid v_1 \mid \square \rangle_{\triangleright}$ or $[c \mid v_1 \mid \square]_{\triangleleft}$ is finite. And by Theorem 2.10 no state repeats, hence the evaluation must eventually reach a stuck state σ . By Lemma 3.4, either $\sigma = [c \mid v_2 \mid \square]_{\triangleright}$ or $\sigma = \langle c \mid v_2 \mid \square \rangle_{\triangleleft}$. \square

Note that, the theorem does not imply that there do not exist machine states from which evaluation will diverge. For example, evaluation starting from $\langle \varepsilon_+ \mid \text{inj}_1 \text{ tt} \mid (\eta_+ \circledast \square) \bullet \square \rangle_{\triangleright}$ will loop forever. This state is, however, not reachable from an initial state because there are no inhabitants of $\mathbb{0}$.

Since the machine is a reversible abstract machine, combinators can also be evaluated backwards:

Definition 3.7 (Π^- -backward evaluation). *The backward evaluation of $c : A \leftrightarrow B$ is*

$eval^{-\dagger}(c) : (B^{\rightarrow}, A^{\leftarrow}) \rightarrow (A^{\rightarrow}, B^{\leftarrow})$ where

$$eval^{-\dagger}(c)(v : B^{\rightarrow}) = \begin{cases} w : A^{\rightarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^{\dagger*} [c \mid w \mid \square]_{\triangleright} \\ w : B^{\leftarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^{\dagger*} \langle c \mid w \mid \square \rangle_{\triangleleft} \end{cases}$$

$$eval^{-\dagger}(c)(v : A^{\leftarrow}) = \begin{cases} w : A^{\rightarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^{\dagger*} [c \mid w \mid \square]_{\triangleright} \\ w : B^{\leftarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^{\dagger*} \langle c \mid w \mid \square \rangle_{\triangleleft} \end{cases}$$

And the forward and backward evaluation functions are inverses of each other, hence both are reversible total functions.

Theorem 3.8 (Π^- -reversible). *$\mathcal{U}_{\text{Agda}}$ For all $c : A \leftrightarrow B$, $V : (A^{\rightarrow}, B^{\leftarrow})$, and $W : (B^{\rightarrow}, A^{\leftarrow})$, we have $eval^{-\dagger}(c)(V) = W$ iff $eval^{-\dagger}(c)(W) = V$.*

Proof. To prove this, an additional lemma is needed:

Lemma 3.9. *$\mathcal{U}_{\text{Agda}}$ For all states σ and σ' , if $\sigma \mapsto^* \sigma'$ then $flip(\sigma') \mapsto^* flip(\sigma)$ where*

$$flip(\sigma) = \begin{cases} \langle c \mid v \mid \kappa \rangle_{\triangleleft} & \text{if } \sigma = \langle c \mid v \mid \kappa \rangle_{\triangleright} \\ \langle c \mid v \mid \kappa \rangle_{\triangleright} & \text{if } \sigma = \langle c \mid v \mid \kappa \rangle_{\triangleleft} \\ [c \mid v \mid \kappa]_{\triangleleft} & \text{if } \sigma = [c \mid v \mid \kappa]_{\triangleright} \\ [c \mid v \mid \kappa]_{\triangleright} & \text{if } \sigma = [c \mid v \mid \kappa]_{\triangleleft} \end{cases}$$

Proof. By induction on the length of $\sigma \mapsto^* \sigma'$. \square

For all $V : (A^{\rightarrow}, B^{\leftarrow})$ and $W : (B^{\rightarrow}, A^{\leftarrow})$, there are four cases:

- $V = v^\rightarrow$ and $W = w^\rightarrow$: If $eval^-(c, V) = W$ then $\langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c \mid w \mid \square]_{\triangleright}$. By Lemma 3.9, $[c \mid w \mid \square]_{\triangleleft} \mapsto^* \langle c \mid v \mid \square \rangle_{\triangleleft}$, hence $eval^{-\dagger}(c)(W) = V$. The other direction is similar.
- $V = v^\rightarrow$ and $W = w^\leftarrow$: Similar.
- $V = v^\leftarrow$ and $W = w^\rightarrow$: Similar.
- $V = v^\leftarrow$ and $W = w^\leftarrow$: Similar.

□

3.3 BIG-STEP INTERPRETER OF Π^-

In this section, a more efficient and more readable specification of a big-step interpreter is described. The interpreter has the same signature as the evaluation relation in Def. 3.5:

$$interp^- : (A \leftrightarrow B) \rightarrow \langle A^\rightarrow, B^\leftarrow \rangle \rightarrow \langle B^\rightarrow, A^\leftarrow \rangle$$

It may be called with a forward or backward value and may return a forward or backward value making the specification somewhat long. The general structure is however simple:

- For base combinators, evaluation in each direction is identical to evaluation in the Π -interpreter: $interp^-(c)(v^\rightarrow) \Downarrow \delta(c, v)^\rightarrow$ and $interp^-(c)(v^\leftarrow) \Downarrow \delta^\dagger(c, v)^\leftarrow$.
- For \oplus there are four cases, the forward cases are identical to the corresponding cases in the Π -interpreter:

$$\frac{interp^-(c_1)(v^\rightarrow) \Downarrow w^\rightarrow}{interp^-(c_1 \oplus c_2)((inj_1 v)^\rightarrow) \Downarrow (inj_1 w)^\rightarrow} \qquad \frac{interp^-(c_1)(v^\rightarrow) \Downarrow w^\rightarrow}{interp^-(c_1 \oplus c_2)((inj_2 v)^\rightarrow) \Downarrow (inj_2 w)^\rightarrow}$$

And the backward cases are as follows:

$$\frac{interp^-(c_1)(v^\leftarrow) \Downarrow w^\leftarrow}{interp^-(c_1 \oplus c_2)((inj_1 v)^\leftarrow) \Downarrow (inj_1 w)^\leftarrow} \qquad \frac{interp^-(c_1)(v^\leftarrow) \Downarrow w^\leftarrow}{interp^-(c_1 \oplus c_2)((inj_2 v)^\leftarrow) \Downarrow (inj_2 w)^\leftarrow}$$

- For \otimes , in the forward direction the evaluation starts from evaluating the first component of the pair. The value of this first component may either be in the same direction as the incoming pair in which case the evaluation continues with the second

component as usual. If however the value of the first component is tagged with the backward direction, the whole evaluation is completed and immediately returns the result:

$$\frac{\text{interp}^-(c_1)(v_1^\rightarrow) \Downarrow w_1^\rightarrow \quad \text{interp}^-(c_2)(v_2^\rightarrow) \Downarrow w_2^\rightarrow}{\text{interp}^-(c_1 \otimes c_2)((v_1, v_2)^\rightarrow) \Downarrow (w_1, w_2)^\rightarrow} \quad \frac{\text{interp}^-(c_1)(v_1^\rightarrow) \Downarrow w^\leftarrow}{\text{interp}^-(c_1 \otimes c_2)((v_1, v_2)^\rightarrow) \Downarrow (w, v_2)^\leftarrow}$$

And the backward evaluation starts from the second component:

$$\frac{\text{interp}^-(c_2)(v_2^\leftarrow) \Downarrow w_2^\leftarrow \quad \text{interp}^-(c_1)(v_1^\leftarrow) \Downarrow w_1^\leftarrow}{\text{interp}^-(c_1 \otimes c_2)((v_1, v_2)^\leftarrow) \Downarrow (w_1, w_2)^\leftarrow} \quad \frac{\text{interp}^-(c_2)(v_2^\leftarrow) \Downarrow w^\rightarrow}{\text{interp}^-(c_1 \otimes c_2)((v_1, v_2)^\leftarrow) \Downarrow (v_1, w)^\rightarrow}$$

- The case for sequential composition $c_1 \mathbin{\text{\$}} c_2$ is the most interesting one. If the incoming value is v^\rightarrow , the evaluation first applies c_1 to v^\rightarrow ; if the result is tagged in the backward direction then the result is returned immediately.

$$\frac{\text{interp}^-(c_1)(v^\rightarrow) \Downarrow w^\leftarrow}{\text{interp}^-(c_1 \mathbin{\text{\$}} c_2)(v^\rightarrow) \Downarrow w^\leftarrow}$$

Otherwise, the evaluation tries to apply c_2 to the result from c_1 . However, the result c_2 might be in the backward direction. To prepare for the possibility that the evaluation goes backward, a handler is needed. The handler $\text{handle}\triangleright: (A \leftrightarrow B) \rightarrow B \rightarrow (B \leftrightarrow C) \rightarrow (C^\rightarrow, A^\leftarrow)$ describes a state in which the intermediate value between c_1 and c_2 is known and evaluation is about to start applying c_2 in the forward direction.

$$\frac{\text{interp}^-(c_1)(v^\rightarrow) \Downarrow w^\rightarrow \quad \text{handle}\triangleright(c_1, w^\rightarrow, c_2) \Downarrow V}{\text{interp}^-(c_1 \mathbin{\text{\$}} c_2)(v^\rightarrow) \Downarrow V}$$

If the result of c_2 is also in the forward direction, the handler returns the result, and the evaluation of the entire sequential composition terminates.

$$\frac{\text{interp}^-(c_2)(v^\rightarrow) \Downarrow w^\rightarrow}{\text{handle}\triangleright(c_1, v^\rightarrow, c_2) \Downarrow w^\rightarrow}$$

If however, the result of evaluating c_2 produces a value flowing in the backward di-

rection, that result needs to be fed into c_1 in the backward direction. To prepare for the possibility that c_1 might produce a value that makes the evaluation goes to the state in between c_1 and c_2 again, yet another handler $handle\triangleleft: (A \leftrightarrow B) \rightarrow B \rightarrow (B \leftrightarrow C) \rightarrow (C^\rightarrow, A^\leftarrow)$ is needed. This handler however describes a state in which the intermediate value between c_1 and c_2 is known but the evaluation is about to start with c_1 in the backward direction.

$$\frac{interp^-(c_2)(v^\rightarrow) \Downarrow w^\leftarrow \quad \text{handle}\triangleleft(c_1, w^\leftarrow, c_2) \Downarrow V}{\text{handle}\triangleright(c_1, v^\rightarrow, c_2) \Downarrow V}$$

If the result of applying c_1 is backward, the whole computation terminates.

$$\frac{interp^-(c_1)(v^\leftarrow) \Downarrow w^\leftarrow}{\text{handle}\triangleleft(c_1, v^\leftarrow, c_2) \Downarrow w^\leftarrow}$$

Otherwise, the evaluation goes to the state in between c_1 and c_2 again, and $handle\triangleright$ can be used in this case.

$$\frac{interp^-(c_1)(v^\leftarrow) \Downarrow w^\rightarrow \quad \text{handle}\triangleright(c_1, w^\rightarrow, c_2) \Downarrow V}{\text{handle}\triangleleft(c_1, v^\leftarrow, c_2) \Downarrow V}$$

Using these two handlers, the backward evaluation can be easily implemented:

$$\frac{interp^-(c_2)(v^\leftarrow) \Downarrow w^\rightarrow}{interp^-(c_1 \ ; \ c_2)(v^\leftarrow) \Downarrow w^\rightarrow} \quad \frac{interp^-(c_2)(v^\leftarrow) \Downarrow w^\leftarrow \quad \text{handle}\triangleleft(c_1, w^\leftarrow, c_2) \Downarrow V}{interp^-(c_1 \ ; \ c_2)(v^\leftarrow) \Downarrow V}$$

The flow of values in the case of sequential composition is intuitively described by the H-shape diagram in Fig. 3.5. Note that this is the same flow of values that happen in both the geometry of interaction [4] and the Int-construction [28]. A priori, there is no guarantee that evaluation will not bounce back and forth in an infinite loop but Theorem 3.10 guarantees termination.

- There are new cases for η_+ that only accept values flowing backwards and flip the

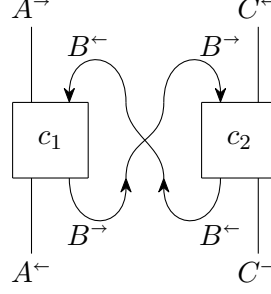


Figure 3.5: Π^- composition

evaluation direction:

$$\text{interp}^-(\eta_+)(\text{inj}_1 v^\leftarrow) \Downarrow \text{inj}_2 (-v)^\rightarrow \qquad \text{interp}^-(\eta_+)(\text{inj}_2 (-v)^\leftarrow) \Downarrow \text{inj}_1 v^\rightarrow$$

- There are new cases for ε_+ that only accept values flowing forward and flip the evaluation direction:

$$\text{interp}^-(\varepsilon_+)(\text{inj}_1 v^\rightarrow) \Downarrow \text{inj}_2 (-v)^\leftarrow \qquad \text{interp}^-(\varepsilon_+)(\text{inj}_2 (-v)^\rightarrow) \Downarrow \text{inj}_1 v^\leftarrow$$

Theorem 3.10. $\mathcal{U}_{\text{Agda}}$ For all c and V , $\text{interp}^-(c)(V) \Downarrow \text{eval}^-(c)(V)$.

Proof. By induction on c :

- c is a base combinator or $\text{id}\leftrightarrow$: By analyzing all possible cases.
- $c = c_1 \otimes c_2$: By analyzing all possible cases and the inductive hypothesis.
- $c = c_1 \oplus c_2$: By analyzing all possible cases and the inductive hypothesis.
- $c = c_1 \circledast c_2$ where $c_1 : A \leftrightarrow B$ and $c_2 : B \leftrightarrow C$: There are two cases:
 - $V = v^\rightarrow$: If $\text{eval}^-(c_1)(v^\rightarrow) = w^\leftarrow$, then by the inductive hypothesis $\text{interp}^-(c_1)(v^\rightarrow) \Downarrow w^\leftarrow$. Hence $\text{eval}^-(c_1 \circledast c_2)(v^\rightarrow) = w^\leftarrow$ and $\text{interp}^-(c_1 \circledast c_2)(v^\rightarrow) \Downarrow w^\leftarrow$. Otherwise, $\text{eval}^-(c_1)(v^\rightarrow) = w^\rightarrow$ and there are two possible cases of $\text{eval}^-(c_1 \circledast c_2)(v^\rightarrow)$:
 - * $\text{eval}^-(c_1 \circledast c_2)(v^\rightarrow) = u^\rightarrow$: An additional lemma is needed for this case.

Lemma 3.11. $\mathcal{U}_{\text{Agda}}$ For any $n \in \mathbb{N}$ and $b : B$, the following hold:

1. If $[c_1 \mid b \mid (\square \circledast c_2) \bullet \square]_{\triangleright} \mapsto^n [c_1 \circledast c_2 \mid u \mid \square]_{\triangleright}$ then $\text{handle}\triangleright(c_1, b^\rightarrow, c_2) \Downarrow u^\rightarrow$.

2. If $\langle c_2 \mid b \mid (c_1 \circledast \square) \bullet \square \rangle_{\triangleleft} \mapsto^n [c_1 \circledast c_2 \mid u \mid \square]_{\triangleright}$ then $\text{handle}\triangleleft(c_1, b^\leftarrow, c_2) \Downarrow u^\rightarrow$.

Proof. By induction on n :

· $n = 0$: No such execution exists, so it is vacuously true.

· $n > 0$:

1. Assume $[c_1 \mid b \mid (\square \dot{\circ} c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \dot{\circ} c_2 \mid u \mid \square]_{\triangleright}$ and its length is n .

There are two possible outcomes of $eval^-(c_2)(b^\rightarrow)$, if $eval^-(c_2)(v^\rightarrow) = x^\rightarrow$ then

$$[c_1 \mid b \mid (\square \dot{\circ} c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \dot{\circ} c_2 \mid x \mid \square]_{\triangleright}$$

Since a Π^- -machine is deterministic so $x = u$ and $interp^-(c_2)(v^\rightarrow) \Downarrow u^\rightarrow$. Hence

$$\frac{interp^-(c_2)(v^\rightarrow) \Downarrow u^\rightarrow}{handle_{\triangleright}(c_1, v^\rightarrow, c_2) \Downarrow u^\rightarrow}$$

If $eval^-(c_2)(v^\rightarrow) = x^\leftarrow$ then

$$[c_1 \mid b \mid (\square \dot{\circ} c_2) \bullet \square]_{\triangleright} \mapsto^* \langle c_2 \mid x \mid (c_1 \dot{\circ} \square) \bullet \square \rangle_{\triangleleft} \mapsto^* [c_1 \dot{\circ} c_2 \mid u \mid \square]_{\triangleright}$$

The length of $\langle c_2 \mid x \mid (c_1 \dot{\circ} \square) \bullet \square \rangle_{\triangleleft} \mapsto^* [c_1 \dot{\circ} c_2 \mid u \mid \square]_{\triangleright}$ is less than n , so by inductive hypothesis $handle_{\triangleleft}(c_1, b^\leftarrow, x) \Downarrow u^\rightarrow$. Therefore

$$\frac{interp^-(c_2)(v^\rightarrow) \Downarrow x^\leftarrow \quad handle_{\triangleleft}(c_1, b^\leftarrow, x) \Downarrow u^\rightarrow}{handle_{\triangleright}(c_1, v^\rightarrow, c_2) \Downarrow u^\rightarrow}$$

2. Similar.

□

Since $eval^-(c_1)(v^\rightarrow) = w^\rightarrow$ so $interp^-(c_1)(v^\rightarrow) \Downarrow w^\rightarrow$ and

$$\langle c_1 \dot{\circ} c_2 \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c_1 \mid w \mid (\square \dot{\circ} c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \dot{\circ} c_2 \mid u \mid \square]_{\triangleright}$$

By Lemma 3.11, $handle_{\triangleright}(c_1, w^\rightarrow, c_2) \Downarrow u^\rightarrow$. Therefore

$$\frac{interp^-(c_1)(v^\rightarrow) \Downarrow w^\rightarrow \quad handle_{\triangleright}(c_1, w^\rightarrow, c_2) \Downarrow u^\rightarrow}{interp^-(c_1 \dot{\circ} c_2)(v^\rightarrow) \Downarrow u^\rightarrow}$$

* $eval^-(c_1 \dot{\circ} c_2)(v^\rightarrow) = u^\leftarrow$: Similar.

– $V = v^\leftarrow$: Similar.

- $c = \varepsilon_+$: By analyzing all possible cases.
- $c = \eta_+$: By analyzing all possible cases.

□

3.4 COMPACT CLOSED CATEGORY

This section will show that Π^- indeed forms a compact closed category.

Theorem 3.12. $\mathcal{U}_{\text{Agda}}$ *The category $(\mathcal{C}, +, \mathbb{0})$ is a compact closed category where*

- $\text{Obj}(\mathcal{C})$ is the set of Π^- types,
- $\text{Hom}(A, B) = \{[c]_{\sim} \mid c : A \leftrightarrow B\}^1$, and
- dual objects $-A$ for every A

Proof. Composition in \mathcal{C} is the concatenation of combinators $_;_$; the identity morphisms are the equivalence classes of $\text{id}\leftrightarrow$ at each type. The basic properties of identity morphisms $\text{eval}^-(\text{id}\leftrightarrow ; c) = \text{eval}^-(c) = \text{eval}^-(c ; \text{id}\leftrightarrow)$, for all c , are straightforward by appealing to the interpreter instead of the abstract machine. The associativity of sequential composition is diagrammatically depicted in Fig. 3.6 where it is intuitively clear that composition is associative.

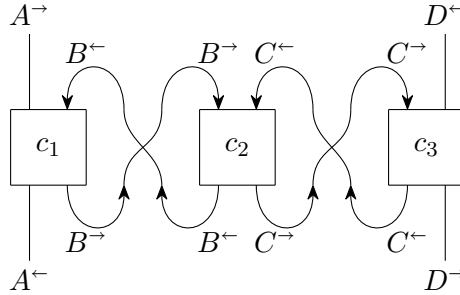


Figure 3.6: Associativity of Π^- composition

The proof of associativity of composition requires a bisimulation between the different execution traces of the abstract machine. There are two possible inputs a^+ and b^- , only a^+ case will be considered here; the other case is similar.

Let $\sigma_0 = \langle c_1 ; (c_2 ; c_3) \mid a \mid \square \rangle_{\triangleright}$ and for each n , let σ_n be the state reachable after n steps $\sigma_0 \mapsto^n \sigma_n$. Define T as follows such that $T(\sigma_n) \mapsto^* T(\sigma_{n+1})$ whenever $\sigma_n \mapsto \sigma_{n+1}$:

¹The definitions of \sim and $[c]_{\sim}$ are the same as in Definition 2.26 and 2.27

$$\begin{aligned}
T(\langle c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid v \mid \square \rangle_d) &= \langle (c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid v \mid \square \rangle_d \\
T(\langle c \mid v \mid \dots (\square \dot{\circ} (c_2 \dot{\circ} c_3)) \bullet \square \rangle_d) &= \langle c \mid v \mid \dots (\square \dot{\circ} c_2) \bullet (\square \dot{\circ} c_3) \bullet \square \rangle_d \\
T(\langle c_2 \dot{\circ} c_3 \mid v \mid (c_1 \dot{\circ} \square) \bullet \square \rangle_d) &= \langle c_2 \mid v \mid (c_1 \dot{\circ} \square) \bullet (\square \dot{\circ} c_3) \bullet \square \rangle_d \\
T(\langle c \mid v \mid \dots (\square \dot{\circ} c_3) \bullet (c_1 \dot{\circ} \square) \bullet \square \rangle_d) &= \langle c \mid v \mid \dots (c_1 \dot{\circ} \square) \bullet (\square \dot{\circ} c_3) \bullet \square \rangle_d \\
T(\langle c \mid v \mid \dots (c_2 \dot{\circ} \square) \bullet (c_1 \dot{\circ} \square) \bullet \square \rangle_d) &= \langle c \mid v \mid \dots ((c_1 \dot{\circ} c_2) \dot{\circ} \square) \bullet \square \rangle_d
\end{aligned}$$

$$\begin{aligned}
T([c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid v \mid \square]_d) &= [(c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid v \mid \square]_d \\
T([c \mid v \mid \dots (\square \dot{\circ} (c_2 \dot{\circ} c_3)) \bullet \square]_d) &= [c \mid v \mid \dots (\square \dot{\circ} c_2) \bullet (\square \dot{\circ} c_3) \bullet \square]_d \\
T([c_2 \dot{\circ} c_3 \mid v \mid (c_1 \dot{\circ} \square) \bullet \square]_d) &= [c_2 \mid v \mid (c_1 \dot{\circ} \square) \bullet (\square \dot{\circ} c_3) \bullet \square]_d \\
T([c \mid v \mid \dots (\square \dot{\circ} c_3) \bullet (c_1 \dot{\circ} \square) \bullet \square]_d) &= [c \mid v \mid \dots (c_1 \dot{\circ} \square) \bullet (\square \dot{\circ} c_3) \bullet \square]_d \\
T([c \mid v \mid \dots (c_2 \dot{\circ} \square) \bullet (c_1 \dot{\circ} \square) \bullet \square]_d) &= [c \mid v \mid \dots ((c_1 \dot{\circ} c_2) \dot{\circ} \square) \bullet \square]_d
\end{aligned}$$

By Lemma 3.4, the transition steps starting from σ_0 end in one of two possible final states.

If:

$$\langle c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid a \mid \square \rangle_{\triangleright} \mapsto^* [c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid v \mid \square]_{\triangleright}$$

the corresponding evaluation on the re-associated combinator can be obtained:

$$T(\sigma_0) = \langle (c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid a \mid \square \rangle_{\triangleright} \mapsto^* [(c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid v \mid \square]_{\triangleright}$$

Otherwise:

$$\langle c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid a \mid \square \rangle_{\triangleright} \mapsto^* \langle c_1 \dot{\circ} (c_2 \dot{\circ} c_3) \mid v \mid \square \rangle_{\triangleleft}$$

therefore:

$$T(\sigma_0) = \langle (c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid a \mid \square \rangle_{\triangleright} \mapsto^* \langle (c_1 \dot{\circ} c_2) \dot{\circ} c_3 \mid v \mid \square \rangle_{\triangleleft}$$

After checking that \mathcal{C} is indeed a category, it remains to verify that $(\mathcal{C}, _+ _)$ is a rigid symmetric monoidal category. It is straightforward to confirm that the right unitors are definable using the left unitors and braiding:

$$\begin{aligned}
&\text{unite}_{+l} : \mathbb{0} + A \leftrightarrow A : \text{uniti}_{+l} \\
&\text{unite}_{+r} = \text{swap}_{+} \dot{\circ} \text{unite}_{+l} : A + \mathbb{0} \leftrightarrow A : \text{uniti}_{+l} \dot{\circ} \text{swap}_{+} = \text{uniti}_{+r}
\end{aligned}$$

All the remaining coherence conditions are straightforward to check (some require bisimulation proofs similar to the above) and are omitted here. \square

Given that Π^- is a reversible language, one might wonder if the category above forms a groupoid where every morphism f has an inverse f^{-1} such that $f \circledast f^{-1} \sim f^{-1} \circledast f \sim \text{id} \leftrightarrow$ like Π . Unfortunately, this is not the case as there is no combinator c such that $\varepsilon_+ \circledast c \sim \text{id} \leftrightarrow$. But for each morphism f there exist morphism f^{-1} such that $f \circledast f^{-1} \circledast f \sim f$ and $f^{-1} \circledast f \circledast f^{-1} \sim f^{-1}$:

Theorem 3.13. $\mathcal{U}_{\text{Agda}}$ For any morphism f in The category \mathcal{C} defined in Thm. 3.12, there exists f^{-1} such that $f \circledast f^{-1} \circledast f \sim f$ and $f^{-1} \circledast f \circledast f^{-1} \sim f^{-1}$.

Proof. To prove this, the definition of the inverse of each combinator is needed. However, the syntactical inverse in [12] does not work because $_ \otimes _$ is not symmetric in Π^- . The inverse that does work is as following:

$$\begin{aligned}
!c &= c' && \text{if } c \text{ is base combinator, where } c' \text{ is } c\text{'s dual} \\
!\text{id} \leftrightarrow &= \text{id} \leftrightarrow \\
!(c_1 \oplus c_2) &= !c_1 \oplus !c_2 \\
!(c_1 \otimes c_2) &= \text{swap}_* \circledast (!c_2 \otimes !c_1) \circledast \text{swap}_* \\
!(c_1 \circledast c_2) &= !c_2 \circledast !c_1 \\
!\eta_+ &= \varepsilon_+ \\
!\varepsilon_+ &= \eta_+
\end{aligned}$$

Two lemmas about $!$ are needed:

Lemma 3.14. $\mathcal{U}_{\text{Agda}}$ For all c , $c \sim !(!c)$.

Proof. By induction on c . □

Lemma 3.15. $\mathcal{U}_{\text{Agda}}$ For all v_1 and v_2 , $\text{eval}^-(c)(v_1) = v_2$ iff $\text{eval}^-(!c)(v_2) = v_1$.

Proof. By induction on c . □

Two things need to be proved:

1. $c \circledast !c \circledast c \sim c$: for all V , there are two cases, only $V = v^\rightarrow$ will be considered here, the other case is similar. There are two possible outcomes of $\text{eval}^-(c)(v)$:

- $eval^-(c)(v) = w^\rightarrow$: By Theorem 3.10 and Lemma 3.15, $interp^-(c)(v^\rightarrow) \Downarrow w^\rightarrow$ and $interp^-(!c)(w^\rightarrow) \Downarrow v^\rightarrow$. Hence

$$\frac{interp^-(c)(v^\rightarrow) \Downarrow w^\rightarrow}{\frac{interp^-(!c)(w^\rightarrow) \Downarrow v^\rightarrow \quad \frac{interp^-(c)(v^\rightarrow) \Downarrow w^\rightarrow}{handle\triangleright(!c, v^\rightarrow, c) \Downarrow w^\rightarrow}}{\frac{interp^-(!c \circ c)(w^\rightarrow) \Downarrow w^\rightarrow}{handle\triangleright(c, w^\rightarrow, !c \circ c) \Downarrow w^\rightarrow}}}$$

- $eval^-(c)(v) = w^\leftarrow$: By Theorem 3.10 and Lemma 3.15, $interp^-(c)(v^\rightarrow) \Downarrow w^\leftarrow$. Hence

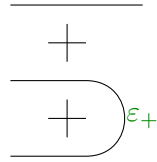
$$\frac{interp^-(c)(v^\rightarrow) \Downarrow w^\leftarrow}{interp^-(c \circ !c)(v^\rightarrow) \Downarrow w^\leftarrow}$$

2. $!c \circ c \circ !c \sim !c$: Using 1. and Lemma 3.14:

$$!c \circ c \circ !c \sim !c \circ !(c) \circ !c \sim !c$$

□

However, the inverse is not unique so it is not an *inverse category* [17, 29]. Consider the combinator $h = id \leftrightarrow \oplus \varepsilon_+$:



There are two combinators that satisfy the conditions:

- $h_1 = id \leftrightarrow \oplus \eta_+$: See Fig.3.7.
- $h_2 = (id \leftrightarrow \oplus \eta_+) \circ A + [B + C] = C + [B + A]$: See Fig.3.8.

Obviously, $id \leftrightarrow \oplus \eta_+ \not\sim (id \leftrightarrow \oplus \eta_+) \circ A + [B + C] = C + [B + A]$.

An immediate consequence of the construction of a compact closed category is that we can define internal hom objects. Specifically, we get a bijection between $A \leftrightarrow B$ and $0 \leftrightarrow (-A + B)$ allowing any combinator $A \leftrightarrow B$ to be used as an object converting demands for values of type A in the backward direction to productions of values of type B in the forward direction. The internal hom is not the familiar exponential object because the

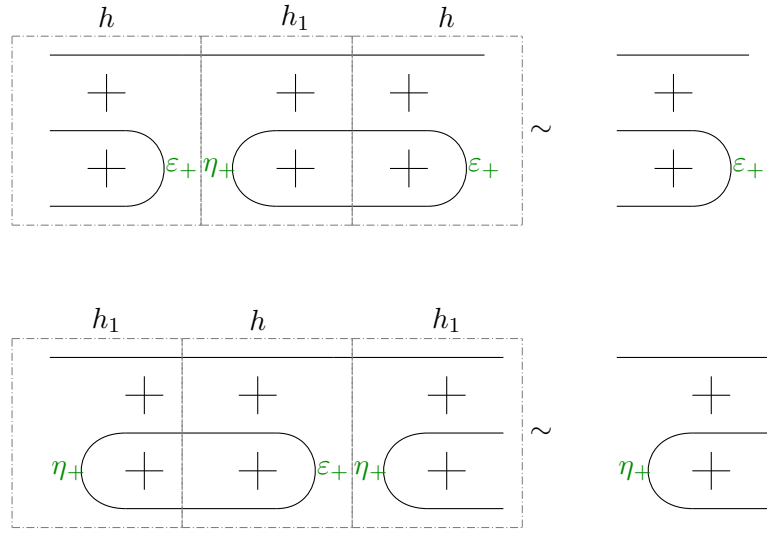


Figure 3.7: $\text{id} \leftrightarrow \oplus \eta_+$ satisfies the inverse conditions for h .

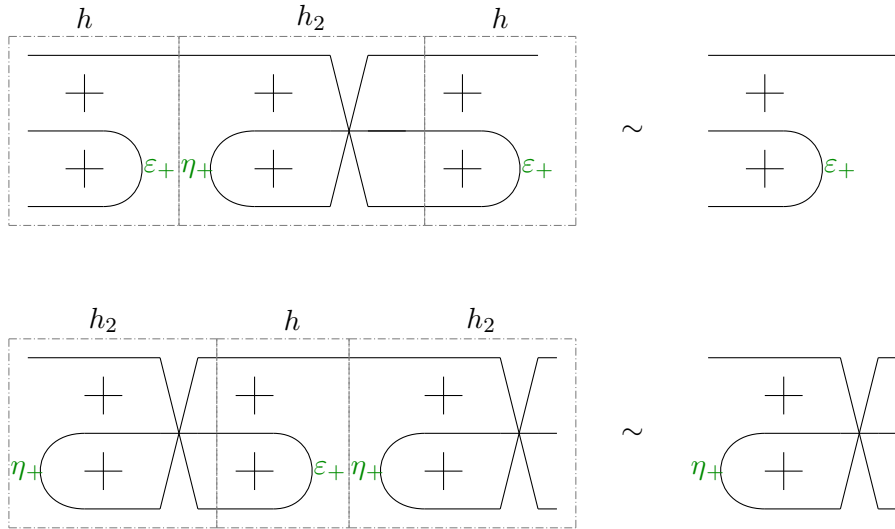


Figure 3.8: $(\text{id} \leftrightarrow \oplus \eta_+); A+[B+C]=C+[B+A]$ satisfies the inverse conditions for h .

relevant tensor is $+$ not \times . In this context, the evaluation map is a combinator of type $(-A + B) + A \leftrightarrow B$ and “currying” converts the combinator on the left to the one on the right in the diagrams below:

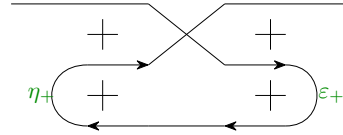


Operationally, the significance of this “currying” is as follows. On the left, f expects either a value of type A or a value of type B . The caller decides. On the right f speculatively opts to receive only values of type A . If the caller provides a value of type A , then f produces a value of type C as before. A value of type B can still be supplied but after backtracking and entering f in the backward direction.

3.5 EXAMPLES

This section will introduce some example Π^- programs, which demonstrate the usage of negative types. The first example is zigzag:

$$\begin{aligned}
 \text{zigzag} &: \mathbb{B} \leftrightarrow \mathbb{B} \\
 \text{zigzag} &= \text{unit}_{+!} \ ; \ (\eta_+ \oplus \text{id} \leftrightarrow) \ ; \\
 & \quad [A+B]+C=[C+B]+A \ ; \\
 & \quad (\varepsilon_+ \oplus \text{id} \leftrightarrow) \ ; \ \text{unite}_{+!}
 \end{aligned}$$



Execution starts in the forward direction. When $\eta_+ \oplus \text{id} \leftrightarrow$ is first encountered, the evaluation proceeds to the right following $\text{id} \leftrightarrow$ until it reaches $\varepsilon_+ \oplus \text{id} \leftrightarrow$. At that point, the evaluation reverses with a negative value. Eventually, the evaluation reaches $\eta_+ \oplus \text{id} \leftrightarrow$ again but this time in the reverse direction. The evaluation then flips direction again and terminates with the same input we started with. Indeed this circuit implements one of the coherence conditions for compact closed categories which essentially states that η_+ following by ε_+ is the identity. Since Π^- forms a compact closed category, as explained in Section 3.4, a combinator can be converted to a higher-order value, and then these values can be composed, curried, and applied:

$$_-\circ_+_ : (A B : \mathbb{U}) \rightarrow \mathbb{U}$$

$$A _-\circ_+ B = - A +_u B$$

$$\text{hof-} : \{A B : \mathbb{U}\} \rightarrow (A \leftrightarrow B) \rightarrow (\mathbb{0} \leftrightarrow A _-\circ_+ B)$$

$$\text{hof- } c = \eta_+ \ ; (c \oplus \text{id}\leftrightarrow) \ ; \text{swap}_+$$

$$\text{comp-} : \{A B C : \mathbb{U}\} \rightarrow (A _-\circ_+ B) +_u (B _-\circ_+ C) \leftrightarrow (A _-\circ_+ C)$$

$$\text{comp-} = \text{assocl}_+ \ ; (\text{assocr}_+ \oplus \text{id}\leftrightarrow) \ ; ((\text{id}\leftrightarrow \oplus \varepsilon_+) \oplus \text{id}\leftrightarrow) \ ; (\text{unite}_{+r} \oplus \text{id}\leftrightarrow)$$

$$\text{curry-} : \{A B C : \mathbb{U}\} \rightarrow (A +_u B \leftrightarrow C) \rightarrow (A \leftrightarrow B _-\circ_+ C)$$

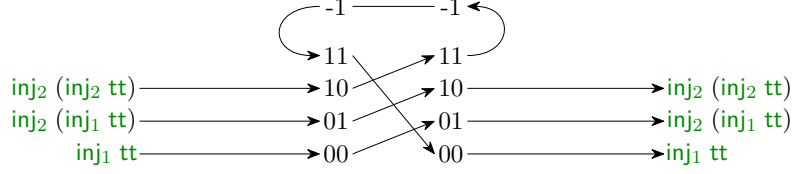
$$\text{curry- } c = \text{uniti}_{+l} \ ; (\eta_+ \oplus \text{id}\leftrightarrow) \ ; \text{swap}_+ \ ; \text{assocl}_+ \ ; (c \oplus \text{id}\leftrightarrow) \ ; \text{swap}_+$$

$$\text{app-} : \{A B : \mathbb{U}\} \rightarrow (A _-\circ_+ B) +_u A \leftrightarrow B$$

$$\text{app-} = \text{swap}_+ \ ; \text{assocl}_+ \ ; (\varepsilon_+ \oplus \text{id}\leftrightarrow) \ ; \text{unite}_{+l}$$

Negative types can also be used to build expressive and more efficient data structures. Consider an enumeration consisting of exactly 2047 elements. Two extreme representations would be to use an inefficient “wide” sum type: or a product type of 11 booleans with some informal convention that one of the elements is unused. With negative types, we can use a type “ $2^{11} - 1$ ” that combines the efficient product type while at the same time enforcing the convention that one of the elements is unused. The advantage of the “ $2^{11} - 1$ ” representation over the wide sum type becomes apparent when writing functions that manipulate either type. For example, to increment a value represented in the additive representation takes 32721 machine transition steps whereas the same operation over the “ $2^{11} - 1$ ” representation only takes 3453 steps.

To more clearly see the role that -1 plays in this setting, let’s consider a smaller example that only contain three elements ($\mathbb{1} + \mathbb{1} + \mathbb{1}$). This can also be represented by a pair of booleans (representing values 00, 01, 10, and 11) minus one element (11) leaving only three values (00, 01, and 10) in the type. The goal is to write a function that increments the three values mapping 00 \mapsto 01, 01 \mapsto 10, and 10 \mapsto 00. This can be achieved using $\text{INCR} : \mathbb{B}^2 \leftrightarrow \mathbb{B}^2$ as shown in the diagram below:

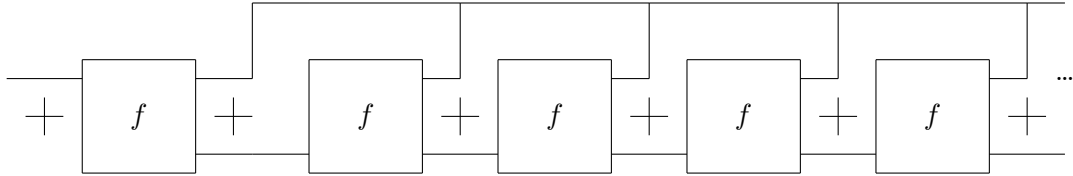


When given the inputs 00 and 01, **INCR** performs the desired action. But when given input 10, **INCR** produces the excluded element 11: this triggers a backtracking action feeding the value 11 back into **INCR** producing 00 as the final result.²

The final example is an implementation of a **for**-loop. Since Π^- forms a compact closed category, so a trace operator can be defined:

$$\begin{aligned}
 \text{trace}_+ &: \forall \{A B C\} \rightarrow (A +_u C \leftrightarrow B +_u C) \rightarrow A \leftrightarrow B \\
 \text{trace}_+ f &= \text{unite}_{+r} \ ; \ (\text{id} \leftrightarrow \oplus \ \eta_+) \ ; \\
 &\quad \text{assocr}_{+} \ ; \ (f \oplus \text{id} \leftrightarrow) \ ; \ \text{assocr}_{+} \ ; \\
 &\quad (\text{id} \leftrightarrow \oplus \ \varepsilon_+) \ ; \ \text{unite}_{+r}
 \end{aligned}$$

Given an input a , the execution of $(\text{trace}_+ f)$ starts by applying f to $\text{inj}_1 a$. The result of f can either be of the form $\text{inj}_1 b$ or $\text{inj}_2 c$. In the first case, the evaluation terminates with result b . In the second case, the evaluation proceeds backwards re-entering f with $\text{inj}_2 c$ and repeating the process. It is equivalent to the following circuit:



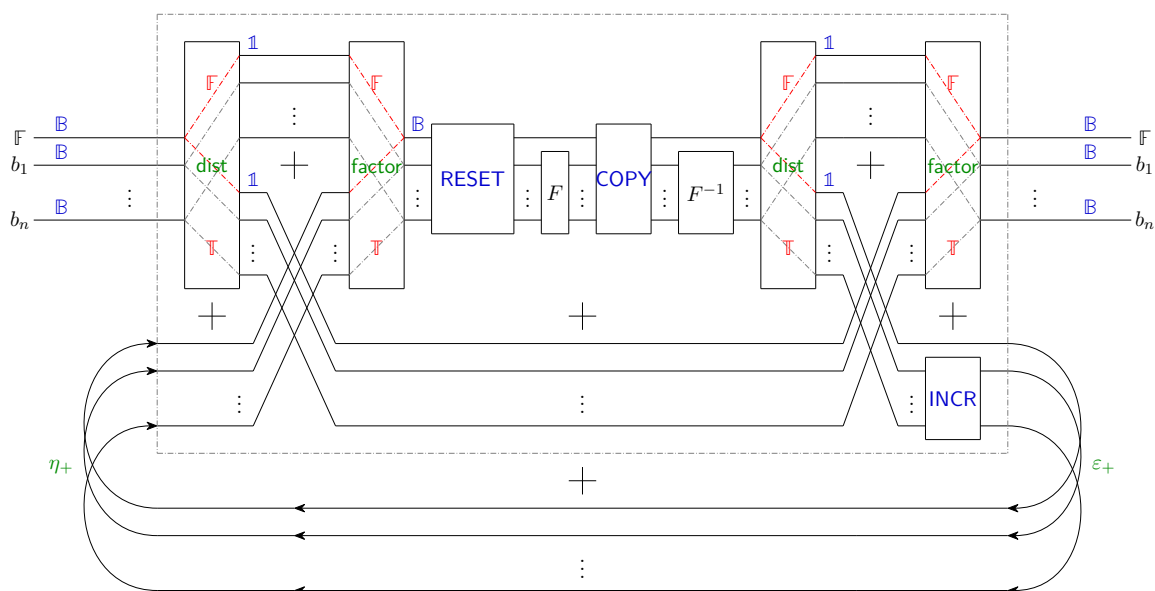
By Theorem 3.6 the execution is guaranteed to terminate, so the number of copies of f is bounded and the trace operator realizes the functionality of a bounded **for**-loop. To demonstrate how to use the trace operator, a combinator that behaves like the following pseudo-code will be presented:

$$\mathbf{for} \ (\bar{b} = \bar{F}; F(\bar{b}) = (\mathbb{T}, _); \bar{b}++)$$

The loop maintains a tuple of booleans representing a binary number that starts at 0 and is

²In general, using negative types, we can build an n -element wrap-around counter using an $n + k$ wrap-around counter for arbitrary k .

incremented at each step using the circuit **INCR**. In each iteration, the current binary number is passed to a function F and the loop terminates when the first bit of F 's return value is \mathbb{T} . The corresponding Π^- program will take the form **LOOP**(F)(\mathbb{F} , \mathbb{F} , \dots , \mathbb{F}) where **LOOP** takes an n -bit reversible function to construct an $n + 1$ -bit reversible function. To maintain reversibility, a common design pattern of reversible programs compute-copy-uncompute [6, 43] is used. In more detail, in each iteration, $F(\bar{b})$ is computed, and the result is copied to an auxiliary wire using **COPY**, and then F will be run backwards to uncompute its result restoring the original \bar{b} . After the copy operation and uncomputing the action of F , the auxiliary wire still holds the copied value and can be used to decide whether to terminate or continue the loop. If the loop continues, we use **RESET** to reset the auxiliary wire to \mathbb{F} in preparation for the next copy operation.



$$\text{LOOP} : \forall \{n\} \rightarrow (\mathbb{B}^{\wedge} n \leftrightarrow \mathbb{B}^{\wedge} n) \rightarrow (\mathbb{B} \times_u \mathbb{B}^{\wedge} n \leftrightarrow \mathbb{B} \times_u \mathbb{B}^{\wedge} n)$$

$$\text{LOOP} \{0\} F = \text{id} \leftrightarrow$$

$$\text{LOOP} \{1\} F = \text{id} \leftrightarrow \otimes F$$

$$\text{LOOP} \{\text{succ}(\text{succ } n)\} F =$$

$$\begin{aligned} & \text{trace}_+ ((\text{dist} \oplus \text{id} \leftrightarrow) ; [A+B]+C=[A+C]+B ; (\text{factor} \oplus \text{id} \leftrightarrow) ; \\ & ((\text{RESET} ; (\text{id} \leftrightarrow \otimes F) ; \text{COPY} ; (\text{id} \leftrightarrow \otimes ! F)) \oplus \text{id} \leftrightarrow) ; \\ & (\text{dist} \oplus \text{id} \leftrightarrow) ; [A+B]+C=[A+C]+B ; (\text{factor} \oplus (\text{id} \leftrightarrow \otimes \text{INCR}))) \end{aligned}$$

Note that, similar to loop unrolling it is possible to build an equivalent circuit in plain Π but at the cost of replicating 2^n copies of the circuit controlled by the trace operator. Negative types enable Π^- to compute iteratively without unrolling the whole computation which leads to exponential size combinators.

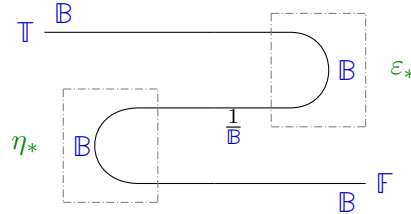
CHAPTER 4

FRACTIONAL TYPES

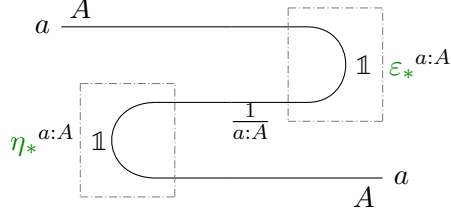
Aiming to construct a compact closed category with multiplicative duals, this chapter will propose a reversible programming language Π' . The analogy to the additive case suggests the introduction of two combinators η_* and ε_* witnessing the isomorphism $\mathbb{1} \leftrightarrow A \times (\mathbb{1}/A)$. The situation however is more subtle than before for at least two reasons. First the empty type $\mathbb{0}$ must be excluded from this extension as otherwise we would have:

$$\begin{aligned} \mathbb{1} &\leftrightarrow \mathbb{0} \times \mathbb{1}/\mathbb{0} && \text{by } \eta_* \\ &\leftrightarrow \mathbb{0} && \text{by } \text{absorbr} \end{aligned}$$

The second subtlety can be appreciated as soon as one attempts to define the operational semantics for η_* . The application of η_* to `tt` at type \mathbb{B} must return a pair whose first component is a boolean, but which one? One approach is to hardwire a default value for each type: η_* will allocate space initialized with hardwired value and a GC token and ε_* will de-allocate them. However, this does not satisfy the snake diagrams. Assume that we hardwire `F` for type \mathbb{B} :



Hence, it is not equivalent to `id`. Inspired by [26], we found that using pointed types can resolve the two issues at once. Instead of introducing η_* and ε_* indexed by types, we introduce $\eta_*^{a:A}$ and $\varepsilon_*^{a:A}$ which are indexed by values. This solves the problem about $\mathbb{1}/\mathbb{0}$ because there is no inhabitant of $\mathbb{0}$, and it satisfies the snake diagrams:



This chapter will propose an extension of Π based on this approach. The structure of this chapter is given as follows:

- Section 4.1 will introduce the syntax of Π' .
- Section 4.2 will introduce the abstract machine semantics of Π' and study its properties.
- Section 4.3 will describe a big-step interpreter of Π' which provides an efficient implementation.
- Section 4.4 will show that the pointed Π' does form a compact closed category.
- Section 4.5 will provide some example Π' programs, which demonstrate the usage of fractional types.

4.1 SYNTAX OF Π'

The syntax of Π' extends Π via adding fractional types and combinators $\eta_*^{a:A}$ and $\varepsilon_*^{a:A}$, and is given in Fig. 4.1.

<i>Value types</i>	$t ::= 0 \mid \mathbb{1} \mid t + t \mid t \times t \mid \frac{\mathbb{1}}{v}$
<i>Values</i>	$v ::= \mathbf{tt} \mid \mathbf{inj}_1 v \mid \mathbf{inj}_2 v \mid (v, v) \mid \circlearrowleft$
<i>Combinator types</i>	$t \leftrightarrow t$
<i>Combinators</i>	$c ::= \mathbf{unite}_+ \mid \mathbf{uniti}_+ \mid \mathbf{swap}_+ \mid \mathbf{assocl}_+ \mid \mathbf{assocr}_+$ $\quad \mid \mathbf{unite}_* \mid \mathbf{uniti}_* \mid \mathbf{swap}_* \mid \mathbf{assocl}_* \mid \mathbf{assocr}_* \mid \mathbf{absorbr} \mid \mathbf{factorzl}$ $\quad \mid \mathbf{dist} \mid \mathbf{factor} \mid \mathbf{id} \leftrightarrow \mid c \circledast c \mid c \oplus c \mid c \otimes c \mid \eta_*^{v:t} \mid \varepsilon_*^{v:t}$
<i>Programs</i>	$p ::= c v$

Figure 4.1: Π' syntax.

The inhabitant of a fractional type $\frac{\mathbb{1}}{A}$ is just a GC token which does not contain any information since all the information is already in its type. The typing judgments of values are given in Fig. 4.2. And the typing judgments of combinators are collected in Fig. 4.3,

where $\eta_*^{a:A}$ and $\varepsilon_*^{a:A}$ have type $\mathbb{1} \leftrightarrow A \times \frac{\mathbb{1}}{a}$. With the two new combinators, the pointed Π' forms a compact closed category. In Sec. 4.2 the operational semantics of Π' will be given and Sec. 4.4 will show that it indeed forms a compact closed category.

$$\frac{}{\mathbf{tt} : \mathbb{1}} \quad \frac{v_1 : t_1 \quad v_2 : t_2}{(v_1, v_2) : t_1 \times t_2} \quad \frac{v_1 : t_1}{\mathbf{inj}_1 v_1 : t_1 + t_2} \quad \frac{v_2 : t_2}{\mathbf{inj}_2 v_2 : t_1 + t_2} \quad \frac{v : t}{\mathbb{C} : \frac{\mathbb{1}}{v}}$$

Figure 4.2: Typing judgments of Π' values.

$$\begin{array}{l} \mathbf{id} \leftrightarrow : \quad t \leftrightarrow t \quad : \mathbf{id} \leftrightarrow \\ \\ \mathbf{unite}_+ | : \quad \mathbb{0} + t \leftrightarrow t \quad : \mathbf{unit}_+ | \\ \mathbf{swap}_+ : \quad t_1 + t_2 \leftrightarrow t_2 + t_1 \quad : \mathbf{swap}_+ \\ \mathbf{assocl}_+ : \quad t_1 + (t_2 + t_3) \leftrightarrow (t_1 + t_2) + t_3 \quad : \mathbf{assocr}_+ \\ \\ \mathbf{unite}_* | : \quad \mathbb{1} \times t \leftrightarrow t \quad : \mathbf{unit}_* | \\ \mathbf{swap}_* : \quad t_1 \times t_2 \leftrightarrow t_2 \times t_1 \quad : \mathbf{swap}_* \\ \mathbf{assocl}_* : \quad t_1 \times (t_2 \times t_3) \leftrightarrow (t_1 \times t_2) \times t_3 \quad : \mathbf{assocr}_* \\ \\ \mathbf{absorbr} : \quad \mathbb{0} \times t \leftrightarrow \mathbb{0} \quad : \mathbf{factorz} | \\ \mathbf{dist} : \quad (t_1 + t_2) \times t_3 \leftrightarrow (t_1 \times t_3) + (t_2 \times t_3) \quad : \mathbf{factor} \\ \\ \eta_*^{a:A} : \quad \mathbb{1} \leftrightarrow A + \frac{\mathbb{1}}{a} \quad : \varepsilon_*^{a:A} \\ \\ \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_2 \leftrightarrow t_3}{\vdash c_1 \mathbin{\&} c_2 : t_1 \leftrightarrow t_3} \quad \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \oplus c_2 : t_1 + t_3 \leftrightarrow t_2 + t_4} \quad \frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \otimes c_2 : t_1 \times t_3 \leftrightarrow t_2 \times t_4} \end{array}$$

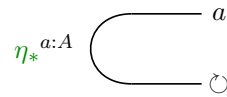
Figure 4.3: Typing judgments of Π' combinators.

4.2 ABSTRACT MACHINE SEMANTICS OF Π'

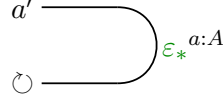
This section will give the abstract machine semantics for Π' . All combinators from Π will still have the same operational semantics as Π ; what's left are the two new combinators $\eta_*^{a:A}$ and $\varepsilon_*^{a:A}$.

As described at the beginning of this chapter:

- $\eta_*^{a:A}$: It will allocate a space for type A initialized to a and a GC token. The GC token does not carry any information because all the information is in its type $\mathbb{C} : \frac{\mathbb{1}}{a}$:



- $\varepsilon_*^{a:A}$: It will deallocate the space and the GC token. However, to maintain the reversibility, it is required to check that if the value has been reset or not:



If $a = a'$ then it will de-allocate both inputs, otherwise it throws an exception.

Since the computation might fail, a failure state need to be added:

Definition 4.1 (Π^d -machine states). *A Π^d -machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa \rangle$ where $c : A \leftrightarrow B$, $v : A$, and $\kappa : \text{CONT}_{A \leftrightarrow B}$.*
- *A return state: $[c \mid v \mid \kappa]$ where $c : A \leftrightarrow B$, $v : B$, and $\kappa : \text{CONT}_{A \leftrightarrow B}$.*
- *A fail state \boxtimes which can make no progress.*

The transition rules for the Π^d -abstract machine are given in Fig. 4.4. They contain all the transition rules for Π and three additional rules for $\eta_*^{a:A}$ and $\varepsilon_*^{a:A}$. In rule \mapsto_{13} a new pair is created with the given value and a GC token. In rules \mapsto_{14} and \mapsto_{15} if the value reaching $\varepsilon_*^{a:A}$ is the expected value then it is collected; otherwise, the machine enters the failure state.

$\langle c \mid v \mid \kappa \rangle$	\mapsto_1	$[c \mid \delta(c, v) \mid \kappa]$	for base combinators c
$\langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle$	\mapsto_2	$[\text{id} \leftrightarrow \mid v \mid \kappa]$	
$\langle c_1 \wp c_2 \mid v \mid \kappa \rangle$	\mapsto_3	$\langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle$	
$\langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle$	\mapsto_4	$\langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle$	
$\langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle$	\mapsto_5	$\langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle$	
$\langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle$	\mapsto_6	$\langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle$	
$[c_1 \mid v \mid (\square \wp c_2) \bullet \kappa]$	\mapsto_7	$\langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle$	
$[c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa]$	\mapsto_8	$\langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle$	
$[c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa]$	\mapsto_9	$[c_1 \otimes c_2 \mid (x, y) \mid \kappa]$	
$[c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa]$	\mapsto_{10}	$[c_1 \wp c_2 \mid v \mid \kappa]$	
$[c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa]$	\mapsto_{11}	$[c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa]$	
$[c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa]$	\mapsto_{12}	$[c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa]$	
$\langle \eta_*^{v:A} \mid \text{tt} \mid \kappa \rangle$	\mapsto_{13}	$[\eta_*^{v:A} \mid (v, \circlearrowleft) \mid \kappa]$	
$\langle \varepsilon_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa \rangle$	\mapsto_{14}	$[\varepsilon_*^{v_1:A} \mid \text{tt} \mid \kappa]$	if $v_1 = v_2$
$\langle \varepsilon_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa \rangle$	\mapsto_{15}	\boxtimes	if $v_1 \neq v_2$

Figure 4.4: Π^d -abstract machine

The machine transition relation is both forward and backward deterministic.

Lemma 4.2 (Π^d -forward deterministic). *$\mathcal{U}Agda$ If $\sigma \mapsto \sigma_1$ and $\sigma \mapsto \sigma_2$ then $\sigma_1 = \sigma_2$*

Proof. By checking all transition rules. \square

Lemma 4.3 (Π' -backward deterministic on proper states). $\mathcal{U}Agda$ *If $\sigma_1 \mapsto \sigma$, $\sigma_2 \mapsto \sigma$, and $\sigma \neq \boxtimes$ then $\sigma_1 = \sigma_2$*

Proof. By checking all transition rules. \square

There is no transition rule from \boxtimes , hence the machine is a partial reversible abstract machine. There are two possibilities for stuck states:

Lemma 4.4 (Stuck). $\mathcal{U}Agda$ *If σ is stuck ($\nexists \sigma'. \sigma \mapsto \sigma'$), then either $\sigma = [c \mid v \mid \square]$ or $\sigma = \boxtimes$.*

Proof. By case analysis on all possible states. \square

So the definition of evaluation for Π' needs to consider the two possible outcomes.

Definition 4.5 (Π' -forward evaluation). *The evaluation of $c : A \leftrightarrow B$ is $eval'(c) : A \rightarrow B \cup \{error\}$ where*

$$eval'(c)(v_1) = \begin{cases} v_2 & \text{if } \langle c \mid v_1 \mid \square \rangle \mapsto^* [c \mid v_2 \mid \square] \\ error & \text{if } \langle c \mid v_1 \mid \square \rangle \mapsto^* \boxtimes \end{cases}$$

Using Lemma 4.4 and the non-repeating theorem for partial reversible abstract machine, the totality of $eval'(c)$ can be proved:

Theorem 4.6 (Π' -termination). $\mathcal{U}Agda$ *For all Π' -combinators $c : A \leftrightarrow B$ and $v_1 : A$, either there exists v_2 such that $eval'(c)(v_1) = v_2$ or $eval'(c)(v_1) = error$.*

Proof. The set of reachable states starting from $\langle c \mid v_1 \mid \square \rangle$ is finite. By Theorem 2.16 no state repeats, hence the evaluation must eventually reach a stuck state σ . By Lemma 4.4, either $\sigma = [c \mid v_2 \mid \square]$ or $\sigma = \boxtimes$. \square

Since the machine is a partial reversible abstract machine, combinators can also be evaluated backwards:

Definition 4.7 (Π' -backward evaluation). *The backward evaluation of $c : A \leftrightarrow B$ is $eval'^{\dagger}(c) : B \rightarrow A \cup \{error\}$ where*

$$eval'^{\dagger}(c, v_1) = \begin{cases} v_2 & \text{if } [c \mid v_1 \mid \square] \mapsto^{\dagger*} \langle c \mid v_2 \mid \square \rangle \\ error & \text{if } [c \mid v_1 \mid \square] \mapsto^{\dagger*} [\eta_*^{v_2:A} \mid (v_3, \circlearrowleft) \mid \kappa] \quad \text{where } v_2 \neq v_3 \end{cases}$$

And the forward and backward evaluation functions are inverses of each other:

Theorem 4.8. $\mathcal{U}Agda$ *For all $c : A \leftrightarrow B$, $v_1 : A$, and $v_2 : B$, we have $eval'(c, v_1) = v_2$ iff $eval'^{\dagger}(c, v_2) = v_1$.*

Proof. If $eval'(c, v_1) = v_2$ then $\langle c \mid v_1 \mid \square \rangle \mapsto^* [c \mid v_2 \mid \square]$. By definition of \mapsto^{\dagger} , we have $[c \mid v_2 \mid \square] \mapsto^{\dagger*} \langle c \mid v_1 \mid \square \rangle$. Hence $eval'^{\dagger}(c, v_2) = v_1$. The other direction is similar. \square

4.3 BIG-STEP INTERPRETER OF Π'

The semantics of Π' is particularly simple to define using a high-level interpreter of type:

$$interp' : (A \leftrightarrow B) \rightarrow A \rightarrow \text{Maybe } B$$

The interpreter is a monadic version of the Π -interpreter, which is given in Fig. 4.5. In the definition, $\gg=$ is the bind operator of maybe monad [8].

$$\begin{aligned} interp'(c)(v) &= \text{just } \delta(c, v) && c \text{ base combinator} \\ interp'(c_1 \oplus c_2)(inj_1 v) &= interp'(c_1)(v) \gg= (\lambda x \rightarrow \text{just } inj_1 x) \\ interp'(c_1 \oplus c_2)(inj_2 v) &= interp'(c_2)(v) \gg= (\lambda x \rightarrow \text{just } inj_2 x) \\ interp'(id \leftrightarrow)(v) &= \text{just } v \\ interp'(c_1 \circledast c_2)(v) &= interp'(c_1)(v) \gg= \\ &\quad (\lambda x \rightarrow interp'(c_2)(x)) \\ interp'(c_1 \otimes c_2)((v_1, v_2)) &= interp'(c_1)(v_1) \gg= \\ &\quad (\lambda x \rightarrow interp'(c_2)(v_2) \gg= \\ &\quad \quad (\lambda y \rightarrow \text{just } (x, y))) \\ interp'(\eta_*^{v:A})(tt) &= \text{just } (v, \circlearrowleft) \\ interp'(\varepsilon_*^{v_1:A})((v_2, \circlearrowleft)) &= \text{just } tt && v_1 = v_2 \\ interp'(\varepsilon_*^{v_1:A})((v_2, \circlearrowleft)) &= \text{nothing} && v_1 \neq v_2 \end{aligned}$$

Figure 4.5: Π' -interpreter

Theorem 4.9 (Π' -interpreter). $\mathcal{U}Agda$ *For all $c : A \leftrightarrow B$, $v_1 : A$, and $v_2 : B$*

- $eval'(c)(v_1) = v_2$ iff $interp'(c)(v_1) = just\ v_2$
- $eval'(c)(v_1) = error$ iff $interp'(c)(v_1) = nothing$.

Proof. By induction on c . □

4.4 COMPACT CLOSED CATEGORY

Since η_* and ε_* are indexed by values, the operational semantics for fractionals does not form a compact closed category directly. Inspired by the relational models of the geometry of interaction [26] we will build the appropriate category (groupoid actually) using pointed types.

Theorem 4.10. *Agda* Let \sim_a be the following equivalence relation on combinators $c_1, c_2 : A \leftrightarrow B$ such that $c_1 \sim c_2$ iff $eval'(c_1)(a) = eval'(c_2)(a)$. \sim_a identifies combinators with the same behavior on a certain point ' a '. We define the category of pointed types $(\mathcal{C}^\bullet, \times, \mathbb{1})$ as follows:

- $Obj(\mathcal{C}^\bullet)$ are of the form (A, a) where A is a Π' -type and $a : A$,
- $Hom((A, a), (B, b)) = [c : A \leftrightarrow B]_{\sim_a}$ where $eval'(c)(a) = b$, and
- dual objects $(\mathbb{1}/a, \circlearrowleft)$ for every (A, a)

The category $(\mathcal{C}^\bullet, \times, \mathbb{1})$ is a compact closed groupoid.

Proof. The composition is the concatenation of combinators $_ \circ _$, this is valid because if $[c_1] \in Hom((A, a), (B, b))$ and $[c_2] \in Hom((B, b), (C, c))$ then we have $eval'(c_1)(a) = b$ and $eval'(c_2)(b) = c$, hence $eval'(c_1 \circ c_2)(a) = c$. And it automatically satisfies the associativity because \sim_a only check the evaluation behavior on point a . For every object (A, a) , we have $[id \leftrightarrow] \in Hom((A, a), (A, a))$ which satisfies the basic properties of identity morphisms.

After checking that \mathcal{C}^\bullet is indeed a category, it remains to verify that $(\mathcal{C}^\bullet, \times, \mathbb{1})$ is a rigid symmetric monoidal category. It is straightforward to check that the right unitors are definable using the left unitors and braiding:

$$\begin{aligned} unite_{*l} & : \mathbb{1} \times A \leftrightarrow A : \quad uniti_{*l} \\ unite_{*r} = swap_* \circ unite_{*l} & : A \times \mathbb{1} \leftrightarrow A : \quad uniti_{*l} \circ swap_* = uniti_{*r} \end{aligned}$$

All the remaining coherence conditions are straightforward to check and are omitted here.

To show that \mathcal{C}^\bullet is a groupoid, the definition of inverse of combinators is needed:

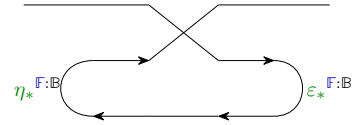
$$\begin{aligned}
!c &= c' && \text{if } c \text{ is base combinator, where } c' \text{ is } c\text{'s dual} \\
!id \leftrightarrow &= id \leftrightarrow \\
!(c_1 \oplus c_2) &= !c_1 \oplus !c_2 \\
!(c_1 \otimes c_2) &= !c_1 \otimes !c_2 \\
!(c_1 \circledast c_2) &= !c_2 \circledast !c_1 \\
!\eta_*^{a:A} &= \varepsilon_*^{a:A} \\
!\varepsilon_*^{a:A} &= \eta_*^{a:A}
\end{aligned}$$

It is straightforward to check that for all $a : A$ and $b : B$, $eval'(c_1)(a) = b$ iff $eval'(!c)(b) = a$. □

4.5 EXAMPLES

This section will introduce some example Π^- programs, which demonstrate the usage of fractional types. The first example is zigzag:

$$\begin{aligned}
\text{zigzag} &: \mathbb{B} \leftrightarrow \mathbb{B} \\
\text{zigzag} &= \text{unit}_*! \circledast (\eta_* \mathbb{F} \otimes id \leftrightarrow) \circledast \\
&[\mathbb{A} \times \mathbb{B}] \times \mathbb{C} = [\mathbb{C} \times \mathbb{B}] \times \mathbb{A} \circledast \\
&(\varepsilon_* \mathbb{F} \otimes id \leftrightarrow) \circledast \text{unite}_*!
\end{aligned}$$



If the input is \mathbb{F} then GC will succeed. Otherwise, it will throw an exception since the value does not match the index of $\varepsilon_*^{\mathbb{F}:\mathbb{B}}$.

Since the pointed $\Pi^/$ forms a compact closed category, as explained in Section 4.4, a combinator can be converted to a higher-order value, and then these values can be composed, curried, and applied.

$$\begin{aligned}
\underline{_} \circ_* \underline{_} &: \{A : \mathbb{U}\} \rightarrow (v : \llbracket A \rrbracket) \rightarrow (B : \mathbb{U}) \rightarrow \mathbb{U} \\
v \circ_* B &= \mathbb{1} / v \times_u B
\end{aligned}$$

hof/ : $\{A B : \mathbb{U}\} \rightarrow (A \leftrightarrow B) \rightarrow (v : \llbracket A \rrbracket) \rightarrow (\mathbb{1} \leftrightarrow v \multimap_* B)$

hof/ $c v = \eta_* v \ ; \ (c \otimes \text{id} \leftrightarrow) \ ; \ \text{swap}_*$

comp/ : $\{A B C : \mathbb{U}\} \{v : \llbracket A \rrbracket\} \rightarrow (w : \llbracket B \rrbracket) \rightarrow (v \multimap_* B) \times_u (w \multimap_* C) \leftrightarrow (v \multimap_* C)$

comp/ $w = \text{assocl}_* \ ; \ (\text{assocr}_* \otimes \text{id} \leftrightarrow) \ ; \ ((\text{id} \leftrightarrow \otimes \varepsilon_* w) \otimes \text{id} \leftrightarrow) \ ; \ (\text{unite}_* \text{r} \otimes \text{id} \leftrightarrow)$

curry/ : $\{A B C : \mathbb{U}\} \rightarrow (A \times_u B \leftrightarrow C) \rightarrow (v : \llbracket B \rrbracket) \rightarrow (A \leftrightarrow v \multimap_* C)$

curry/ $c v = \text{uniti}_* \text{l} \ ; \ (\eta_* v \otimes \text{id} \leftrightarrow) \ ; \ \text{swap}_* \ ; \ \text{assocl}_* \ ; \ (c \otimes \text{id} \leftrightarrow) \ ; \ \text{swap}_*$

app/ : $\{A B : \mathbb{U}\} \rightarrow (v : \llbracket A \rrbracket) \rightarrow (v \multimap_* B) \times_u A \leftrightarrow B$

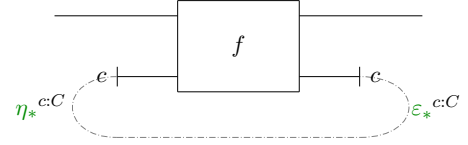
app/ $v = \text{swap}_* \ ; \ \text{assocl}_* \ ; \ (\varepsilon_* _ \otimes \text{id} \leftrightarrow) \ ; \ \text{unite}_* \text{l}$

The most important construction from compact closed categories is trace, which provides a scoped allocation mechanism:

trace_{*} : $\forall \{A B C\} \rightarrow \llbracket C \rrbracket \rightarrow (A \times_u C \leftrightarrow B \times_u C)$

$\rightarrow A \leftrightarrow B$

trace_{*} $v f = \text{uniti}_* \text{r} \ ; \ (\text{id} \leftrightarrow \otimes \eta_* v)$
 $\ ; \ \text{assocl}_* \ ; \ (f \otimes \text{id} \leftrightarrow) \ ; \ \text{assocr}_*$
 $\ ; \ (\text{id} \leftrightarrow \otimes \varepsilon_* v) \ ; \ \text{unite}_* \text{r}$



In the trace, $f : A \times C \leftrightarrow B \times C$ is allowed to use the value $c : C$ during the computation. To maintain reversibility, it is required that f resets the value back to c when the computation is finished, otherwise it will throw an exception. This is similar to:

- Quipper [25]:

```
with_ancilla :: (Qubit -> Circ a) -> Circ a
```

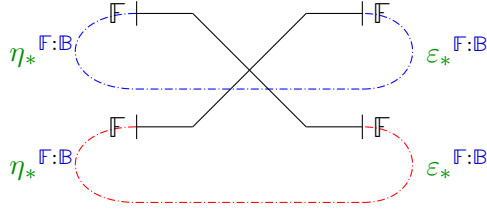
The operator takes a block of gates parameterized by an ancilla value, allocates a new ancilla value of type `Qubit` initialized to $|0\rangle$, and runs the given block of gates. At the end of its execution, the block is expected to return the ancilla value to the state $|0\rangle$ at which point it is de-allocated. The expectation that the ancilla value is in the state $|0\rangle$ is enforced via a runtime check.

- Ricercar [41]: The expression $\alpha x.A$ allocates an ancilla wire x for the gate A requiring that x is set to 0 after the evaluation of A as the following rule of the operational semantics shows:

$$\frac{\sigma \vdash x \rightarrow b \quad \sigma[x \mapsto 0] \vdash A \rightarrow \sigma' \quad \sigma' \vdash x \rightarrow 0}{\sigma \vdash \alpha x.A \rightarrow \sigma'[x \mapsto b]}$$

where σ is the global memory mapping each variable to its value and \rightarrow represents the transition relation.

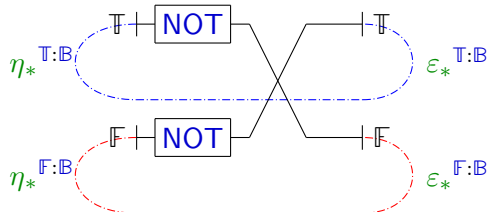
Fractional types provide more flexibility than the scoped mechanism. The GC token can be used whenever the value matches:



Exchange : $\mathbb{1} \leftrightarrow \mathbb{1}$

$$\begin{aligned} \text{Exchange} = & \text{uniti}_{*r} \ ; \ (\text{id} \leftrightarrow \otimes \text{uniti}_{*r}) \ ; \ (\text{id} \leftrightarrow \otimes \eta_* \{B\} \ F \otimes \eta_* \ F) \ ; \\ & (\text{id} \leftrightarrow \otimes [A \times B] \times [C \times D] = [C \times C] \times [A \times D]) \ ; \\ & (\text{id} \leftrightarrow \otimes \varepsilon_* \ F \otimes \varepsilon_* \ F) \ ; \ (\text{id} \leftrightarrow \otimes \text{unite}_{*r}) \ ; \ \text{unite}_{*r} \end{aligned}$$

The upper F is GCed using the GC token generated by the lower $\eta_*^{F:B}$, and the lower F is GCed using the GC token generated by the upper $\eta_*^{F:B}$. And the value does not require to be reset to its initial state; it only requires matching values:



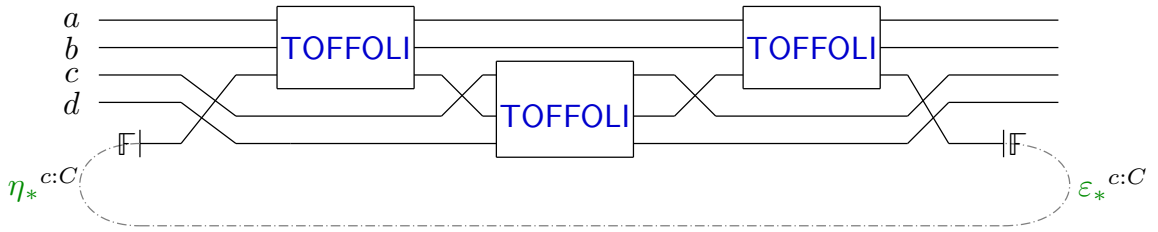
Exchange' : $\mathbb{1} \leftrightarrow \mathbb{1}$

$$\text{Exchange}' = \text{uniti}_{*r} \ ; \ (\text{id} \leftrightarrow \otimes \text{uniti}_{*r}) \ ; \ (\text{id} \leftrightarrow \otimes \eta_* \ T \otimes \eta_* \ F) \ ;$$

$$\begin{aligned}
& (\text{id} \leftrightarrow \otimes (((\text{NOT} \otimes \text{id} \leftrightarrow) \otimes (\text{NOT} \otimes \text{id} \leftrightarrow))) \ ; \\
& [A \times B] \times [C \times D] = [C \times C] \times [A \times D]) \ ; \\
& (\text{id} \leftrightarrow \otimes \varepsilon_* \top \otimes \varepsilon_* \mathbb{F}) \ ; (\text{id} \leftrightarrow \otimes \text{unite}_{*r}) \ ; \text{unite}_{*r}
\end{aligned}$$

The upper \top does not need to be reset back to \top . Instead it is flipped to \mathbb{F} and GCed using the GC token \circlearrowleft : $\frac{1}{\mathbb{F}}$.

An application of fractional types is the construction of 4-bit Toffoli gates from 3-bit Toffoli gates [42]:



$$\text{toffoli4} : \mathbb{B}^4 \leftrightarrow \mathbb{B}^4$$

$$\begin{aligned}
\text{toffoli4} = \text{trace}_* \mathbb{F} & ([A \times B \times C \times D] \times E = [A \times B \times E] \times [C \times D]) \ ; (\text{toffoli} \otimes \text{id} \leftrightarrow) \ ; \\
& [A \times B \times E] \times [C \times D] = [A \times B] \times [C \times E \times D] \ ; \\
& (\text{id} \leftrightarrow \otimes \text{toffoli}) \ ; \\
& ! [A \times B \times E] \times [C \times D] = [A \times B] \times [C \times E \times D] \ ; \\
& (\text{toffoli} \otimes \text{id} \leftrightarrow) \ ; ! [A \times B \times C \times D] \times E = [A \times B \times E] \times [C \times D]
\end{aligned}$$

The ancilla wire e carries $a \text{ xor } b$ after the first Toffli gate ($e = a \text{ xor } b \text{ xor } \mathbb{F} = a \text{ xor } b$), and it is used to compute the result of the 4-bit Toffoli gate ($d = c \text{ xor } e = c \text{ xor } a \text{ xor } b$). The last Toffoli gate resets e back to \mathbb{F} ($e = a \text{ xor } b \text{ xor } (a \text{ xor } b) = \mathbb{F}$). In chapter 5, we will use fractional types to provide enough fresh variables for a SAT solver.

CHAPTER 5

FIELD OF TYPES

In this chapter, we will put negative and fractional types together to form a new language $\Pi^{\mathbb{Q}}$. $\Pi^{\mathbb{Q}}$ allows us to use both effects in our programs. To demonstrate the expressiveness of $\Pi^{\mathbb{Q}}$, we implement a SAT solver in $\Pi^{\mathbb{Q}}$ via exploiting both trace operators from Π^- and Π' .

The structure of this chapter is given as follows:

- Section 5.1 will introduce the syntax of $\Pi^{\mathbb{Q}}$.
- Section 5.2 will introduce the abstract machine semantics of $\Pi^{\mathbb{Q}}$, and study its properties.
- Section 5.3 will describe a big-step interpreter of $\Pi^{\mathbb{Q}}$ which provides an efficient implementation.
- Section 5.4 will provide some examples of $\Pi^{\mathbb{Q}}$ programs and the implementation of a SAT solver.

5.1 SYNTAX OF $\Pi^{\mathbb{Q}}$

The syntax of Π^- extends Π via adding negative and fractional types. Four combinators (η_+ , ε_+ , $\eta_*^{v:t}$ and $\varepsilon_*^{v:t}$) are added; the syntax is summarized in Fig. 5.1.

The typing judgments are the same as in Π^- and Π' and are given in Fig. 5.2 and 5.3.

5.2 ABSTRACT MACHINE SEMANTICS OF $\Pi^{\mathbb{Q}}$

This section will give an abstract machine semantics for $\Pi^{\mathbb{Q}}$. In order to manage negative types, normal machine states need to contain direction. And a failure state is also needed

<i>Value types</i>	$t ::= \mathbb{0} \mid \mathbb{1} \mid t + t \mid t \times t \mid -t \mid \frac{\mathbb{1}}{v}$
<i>Values</i>	$v ::= \mathbf{tt} \mid \mathbf{inj}_1 v \mid \mathbf{inj}_2 v \mid (v, v) \mid -v \mid \circlearrowleft$
<i>Combinator types</i>	$t \leftrightarrow t$
<i>Combinators</i>	$c ::= \mathbf{unite}_+ \mid \mathbf{uniti}_+ \mid \mathbf{swap}_+ \mid \mathbf{assocl}_+ \mid \mathbf{assocr}_+ \mid \mathbf{unite}_* \mid \mathbf{uniti}_* \mid \mathbf{swap}_* \mid \mathbf{assocl}_* \mid \mathbf{assocr}_* \mid \mathbf{absorbr} \mid \mathbf{factorzl} \mid \mathbf{dist} \mid \mathbf{factor} \mid \mathbf{id} \leftrightarrow \mid c \mathbin{\text{\textcircled{;}}} c \mid c \oplus c \mid c \otimes c \mid \eta_+ \mid \varepsilon_+ \mid \eta_*^{v:t} \mid \varepsilon_*^{v:t}$
<i>Programs</i>	$p ::= c v$

Figure 5.1: $\Pi^{\mathbb{Q}}$ syntax.

$\frac{}{\mathbf{tt} : \mathbb{1}}$	$\frac{v_1 : t_1 \quad v_2 : t_2}{(v_1, v_2) : t_1 \times t_2}$	$\frac{v_1 : t_1}{\mathbf{inj}_1 v_1 : t_1 + t_2}$	$\frac{v_2 : t_2}{\mathbf{inj}_2 v_2 : t_1 + t_2}$	$\frac{v : t}{-v : -t}$	$\frac{v : t}{\circlearrowleft : \frac{\mathbb{1}}{v}}$
-------------------------------------	---	--	--	-------------------------	---

Figure 5.2: Typing judgments of $\Pi^{\mathbb{Q}}$ values.

$\mathbf{id} \leftrightarrow :$	$t \leftrightarrow t$	$: \mathbf{id} \leftrightarrow$
$\mathbf{unite}_+ :$	$\mathbb{0} + t \leftrightarrow t$	$: \mathbf{uniti}_+$
$\mathbf{swap}_+ :$	$t_1 + t_2 \leftrightarrow t_2 + t_1$	$: \mathbf{swap}_+$
$\mathbf{assocl}_+ :$	$t_1 + (t_2 + t_3) \leftrightarrow (t_1 + t_2) + t_3$	$: \mathbf{assocr}_+$
$\mathbf{unite}_* :$	$\mathbb{1} \times t \leftrightarrow t$	$: \mathbf{uniti}_*$
$\mathbf{swap}_* :$	$t_1 \times t_2 \leftrightarrow t_2 \times t_1$	$: \mathbf{swap}_*$
$\mathbf{assocl}_* :$	$t_1 \times (t_2 \times t_3) \leftrightarrow (t_1 \times t_2) \times t_3$	$: \mathbf{assocr}_*$
$\mathbf{absorbr} :$	$\mathbb{0} \times t \leftrightarrow \mathbb{0}$	$: \mathbf{factorzl}$
$\mathbf{dist} :$	$(t_1 + t_2) \times t_3 \leftrightarrow (t_1 \times t_3) + (t_2 \times t_3)$	$: \mathbf{factor}$
$\eta_+ :$	$\mathbb{0} \leftrightarrow t + (-t)$	$: \varepsilon_+$
$\eta_*^{a:A} :$	$\mathbb{1} \leftrightarrow A + \frac{\mathbb{1}}{a}$	$: \varepsilon_*^{a:A}$
$\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_2 \leftrightarrow t_3}{\vdash c_1 \mathbin{\text{\textcircled{;}}} c_2 : t_1 \leftrightarrow t_3}$	$\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \oplus c_2 : t_1 + t_3 \leftrightarrow t_2 + t_4}$	$\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \otimes c_2 : t_1 \times t_3 \leftrightarrow t_2 \times t_4}$

Figure 5.3: Typing judgments of $\Pi^{\mathbb{Q}}$ combinators.

for fractional types. The definition of extended machine states is:

Definition 5.1 ($\Pi^{\mathbb{Q}}$ -machine states). *A $\Pi^{\mathbb{Q}}$ -machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa \rangle_d$ where $c : A \leftrightarrow B$, $v : A$, $\kappa : \text{CONT}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*
- *A return state: $[c \mid v \mid \kappa]_d$ where $c : A \leftrightarrow B$, $v : B$, $\kappa : \text{CONT}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*
- *A fail state \boxtimes which can make no progress.*

Fig. 5.4 gives the transition rules of the abstract machine. It contains all of the transition rules from Π , Π^- and Π^{\setminus} . In addition, we need to add backward transition rules for $\eta_*^{v:t}$ and $\varepsilon_*^{v:t}$ (rules 20,21, and 22).

For base combinators c :

$\langle c \mid v \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{1}} [c \mid \delta(c, v) \mid \kappa]_{\triangleright}$ $\langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{2}} [\text{id} \leftrightarrow \mid v \mid \kappa]_{\triangleright}$ $\langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{3}} \langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleright}$ $\langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{4}} \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleright}$ $\langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{5}} \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleright}$ $\langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{6}} \langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleright}$ $\langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{7}} \langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleright}$ $\langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{8}} \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleright}$ $\langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{9}} [c_1 \otimes c_2 \mid (x, y) \mid \kappa]_{\triangleright}$ $\langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{10}} [c_1 \wp c_2 \mid v \mid \kappa]_{\triangleright}$ $\langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{11}} [c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa]_{\triangleright}$ $\langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleright} \xrightarrow{\vec{12}} [c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa]_{\triangleright}$ $\langle \epsilon_+ \mid \text{inj}_1 v \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{13}} \langle \epsilon_+ \mid \text{inj}_2 (-v) \mid \kappa \rangle_{\triangleleft}$ $\langle \epsilon_+ \mid \text{inj}_2 (-v) \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{14}} \langle \epsilon_+ \mid \text{inj}_1 v \mid \kappa \rangle_{\triangleleft}$ $\langle \eta_*^{v:A} \mid \text{tt} \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{17}} [\eta_*^{v:A} \mid (v, \circlearrowleft) \mid \kappa]_{\triangleright}$ $\langle \epsilon_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{18}} [\epsilon_*^{v_1:A} \mid \text{tt} \mid \kappa]_{\triangleright}$ $\langle \epsilon_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa \rangle_{\triangleright} \xrightarrow{\vec{19}} \boxtimes$ $\langle \epsilon_*^{v:A} \mid \text{tt} \mid \kappa \rangle_{\triangleleft} \xrightarrow{\vec{22}} \langle \epsilon_*^{v:A} \mid (v, \circlearrowleft) \mid \kappa \rangle_{\triangleleft}$ $[\eta_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa]_{\triangleleft} \xrightarrow{\vec{20}} \langle \eta_*^{v_1:A} \mid \text{tt} \mid \kappa \rangle_{\triangleleft}$ $[\eta_*^{v_1:A} \mid (v_2, \circlearrowleft) \mid \kappa]_{\triangleleft} \xrightarrow{\vec{21}} \boxtimes$	$[c \mid v \mid \kappa]_{\triangleleft} \xleftarrow{\vec{1}} \langle c \mid \delta^\dagger(c, v) \mid \kappa \rangle_{\triangleleft}$ $[\text{id} \leftrightarrow \mid v \mid \kappa]_{\triangleleft} \xleftarrow{\vec{2}} \langle \text{id} \leftrightarrow \mid v \mid \kappa \rangle_{\triangleleft}$ $\langle c_1 \mid v \mid (\square \wp c_2) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{3}} \langle c_1 \wp c_2 \mid v \mid \kappa \rangle_{\triangleleft}$ $\langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{4}} \langle c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa \rangle_{\triangleleft}$ $\langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{5}} \langle c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa \rangle_{\triangleleft}$ $\langle c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{6}} \langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle_{\triangleleft}$ $\langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{7}} [c_1 \mid v \mid (\square \wp c_2) \bullet \kappa]_{\triangleleft}$ $\langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleleft} \xleftarrow{\vec{8}} [c_1 \mid x \mid (\square \otimes [c_2, y]) \bullet \kappa]_{\triangleleft}$ $[c_1 \otimes c_2 \mid (x, y) \mid \kappa]_{\triangleleft} \xleftarrow{\vec{9}} \langle c_2 \mid y \mid ([c_1, x] \otimes \square) \bullet \kappa \rangle_{\triangleleft}$ $[c_1 \wp c_2 \mid v \mid \kappa]_{\triangleleft} \xleftarrow{\vec{10}} \langle c_2 \mid v \mid (c_1 \wp \square) \bullet \kappa \rangle_{\triangleleft}$ $[c_1 \oplus c_2 \mid \text{inj}_1 x \mid \kappa]_{\triangleleft} \xleftarrow{\vec{11}} \langle c_1 \mid x \mid (\square \oplus c_2) \bullet \kappa \rangle_{\triangleleft}$ $[c_1 \oplus c_2 \mid \text{inj}_2 y \mid \kappa]_{\triangleleft} \xleftarrow{\vec{12}} \langle c_2 \mid y \mid (c_1 \oplus \square) \bullet \kappa \rangle_{\triangleleft}$ $[\eta_+ \mid \text{inj}_2 (-v) \mid \kappa]_{\triangleleft} \xrightarrow{\vec{15}} [\eta_+ \mid \text{inj}_1 v \mid \kappa]_{\triangleright}$ $[\eta_+ \mid \text{inj}_1 v \mid \kappa]_{\triangleleft} \xrightarrow{\vec{16}} [\eta_+ \mid \text{inj}_2 (-v) \mid \kappa]_{\triangleright}$ $\text{if } v_1 = v_2$ $\text{if } v_1 \neq v_2$ $\text{if } v_1 = v_2$ $\text{if } v_1 \neq v_2$
---	---

Figure 5.4: $\Pi^{\mathbb{Q}}$ -abstract machine transition rules

The machine transition relation is both forward and backward deterministic for normal states.

Lemma 5.2 ($\Pi^{\mathbb{Q}}$ -forward deterministic). $\mathcal{U}Agda$ If $\sigma \mapsto \sigma_1$ and $\sigma \mapsto \sigma_2$ then $\sigma_1 = \sigma_2$

Proof. By checking all transitions. □

Lemma 5.3 ($\Pi^{\mathbb{Q}}$ -backward deterministic). $\mathcal{U}Agda$ If $\sigma_1 \mapsto \sigma$, $\sigma_2 \mapsto \sigma$, and $\sigma \neq \boxtimes$ then $\sigma_1 = \sigma_2$

Proof. By checking all transitions. □

There is no transition rule from \boxtimes , hence the machine is a partial reversible abstract machine. There are three possibilities for stuck states:

Lemma 5.4 (Stuck). $\mathcal{U}Agda$ If σ is stuck ($\nexists \sigma'. \sigma \mapsto \sigma'$), then either $\sigma = \langle c \mid v \mid \square \rangle_{\triangleleft}$, $\sigma = [c \mid v \mid \square]_{\triangleright}$, or $\sigma = \boxtimes$.

Proof. By case analysis on all possible states. □

So the definition of evaluation for $\Pi^{\mathbb{Q}}$ needs to consider the three possible outcomes:

Definition 5.5 ($\Pi^{\mathbb{Q}}$ -forward evaluation). *The evaluation of $c : A \leftrightarrow B$ is*

$$eval^{\mathbb{Q}}(c) : (A^{\rightarrow}, B^{\leftarrow}) \rightarrow (B^{\rightarrow}, A^{\leftarrow}) \uplus \{error\}$$

where

$$eval^{\mathbb{Q}}(c, v : A^{\rightarrow}) = \begin{cases} w : B^{\rightarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c \mid w \mid \square]_{\triangleright} \\ w : A^{\leftarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* \langle c \mid w \mid \square \rangle_{\triangleleft} \\ error & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* \boxtimes \end{cases}$$

$$eval^{\mathbb{Q}}(c, v : B^{\leftarrow}) = \begin{cases} w : B^{\rightarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^* [c \mid w \mid \square]_{\triangleright} \\ w : A^{\leftarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^* \langle c \mid w \mid \square \rangle_{\triangleleft} \\ error & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^* \boxtimes \end{cases}$$

Using Lemma 5.4 and the non-repeating theorem for partial reversible abstract machine, the totality of $eval^{\Pi^{\mathbb{Q}}}(c)$ can be proved:

Theorem 5.6 ($\Pi^{\mathbb{Q}}$ -termination). *For all $\Pi^{\mathbb{Q}}$ -combinators $c : A \leftrightarrow B$ and $v_1 : (A^{\rightarrow}, B^{\leftarrow})$ either there exists $v_2 : (B^{\rightarrow}, A^{\leftarrow})$ such that $eval^{\mathbb{Q}}(c)(v_1) = v_2$ or $eval^{\mathbb{Q}}(c)(v_1) = error$.*

Proof. The set of reachable states starting from $\langle c \mid v_1 \mid \square \rangle_{\triangleright}$ and $[c \mid v_1 \mid \square]_{\triangleleft}$ is finite. By Theorem 2.16 no state repeats, hence the evaluation must eventually reach a stuck state σ . By Lemma 5.4, either $\sigma = \langle c \mid v_2 \mid \square \rangle_{\triangleleft}$, $\sigma = [c \mid v_2 \mid \square]_{\triangleright}$, or $\sigma = \boxtimes$. \square

Since the machine is a partial reversible abstract machine, combinators can also be evaluated backward:

Definition 5.7 ($\Pi^{\mathbb{Q}}$ -backward evaluation). *The backward evaluation of $c : A \leftrightarrow B$ is*

$$eval^{\mathbb{Q}}(c) : (B^{\rightarrow}, A^{\leftarrow}) \rightarrow (A^{\rightarrow}, B^{\leftarrow}) \uplus \{error\}$$

where

$$\begin{aligned}
eval^{\mathbb{Q}\dagger}(c, v : B^\rightarrow) &= \begin{cases} w : A^\rightarrow & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^{\dagger*} \langle c \mid w \mid \square \rangle_{\triangleleft} \\ w : B^\leftarrow & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^{\dagger*} [c \mid w \mid \square]_{\triangleright} \\ \text{error} & \text{if } [c \mid v \mid \square]_{\triangleleft} \mapsto^{\dagger*} \boxtimes \end{cases} \\
eval^{\mathbb{Q}\dagger}(c, v : A^\leftarrow) &= \begin{cases} w : A^\rightarrow & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^{\dagger*} \langle c \mid w \mid \square \rangle_{\triangleleft} \\ w : B^\leftarrow & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^{\dagger*} [c \mid w \mid \square]_{\triangleright} \\ \text{error} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^{\dagger*} \boxtimes \end{cases}
\end{aligned}$$

And the forward and backward evaluation functions are inverses of each other:

Theorem 5.8 ($\Pi^{\mathbb{Q}}$ -reversible). $\mathcal{U}Agda$ For all $c : A \leftrightarrow B$, $V : \langle A^\rightarrow, B^\leftarrow \rangle$, and $W : \langle B^\rightarrow, A^\leftarrow \rangle$, we have $eval^{\mathbb{Q}}(c)(V) = W$ iff $eval^{\mathbb{Q}\dagger}(c)(W) = V$.

Proof. To prove this, an additional lemma is needed:

Lemma 5.9. $\mathcal{U}Agda$ For all non-stuck states σ and σ' , if $\sigma \mapsto^* \sigma'$ then $flip(\sigma') \mapsto^* flip(\sigma)$ where

$$flip(\sigma) = \begin{cases} \langle c \mid v \mid \kappa \rangle_{\triangleleft} & \text{if } \sigma = \langle c \mid v \mid \kappa \rangle_{\triangleright} \\ \langle c \mid v \mid \kappa \rangle_{\triangleright} & \text{if } \sigma = \langle c \mid v \mid \kappa \rangle_{\triangleleft} \\ [c \mid v \mid \kappa]_{\triangleleft} & \text{if } \sigma = [c \mid v \mid \kappa]_{\triangleright} \\ [c \mid v \mid \kappa]_{\triangleright} & \text{if } \sigma = [c \mid v \mid \kappa]_{\triangleleft} \end{cases}$$

Proof. By induction on the length of $\sigma \mapsto^* \sigma'$. □

For all $V : \langle A^\rightarrow, B^\leftarrow \rangle$ and $W : \langle B^\rightarrow, A^\leftarrow \rangle$, there are four cases:

- $V = v^\rightarrow$ and $W = w^\rightarrow$: If $eval^{\mathbb{Q}}(c, V) = W$ then $\langle c \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c \mid w \mid \square]_{\triangleright}$. By Lemma 5.9, $[c \mid w \mid \square]_{\triangleleft} \mapsto^* \langle c \mid v \mid \square \rangle_{\triangleleft}$, hence $eval^{\mathbb{Q}\dagger}(c)(W) = V$. The other direction is similar.
- $V = v^\rightarrow$ and $W = w^\leftarrow$: Similar.
- $V = v^\leftarrow$ and $W = w^\rightarrow$: Similar.
- $V = v^\leftarrow$ and $W = w^\leftarrow$: Similar.

□

5.3 BIG-STEP INTERPRETER OF Π^Q

In this section, a more efficient and more readable specification of a big-step interpreter is described. The interpreter has a similar signature as the evaluation relation in Def. 5.5:

$$\text{interp} : (A \leftrightarrow B) \rightarrow \langle A^\rightarrow, B^\leftarrow \rangle \rightarrow \text{Maybe} \langle B^\rightarrow, A^\leftarrow \rangle.$$

The implementation will use both the maybe monad and exception handlers. The forward part is similar to the interpreter of Π' except for combinator composition:

$$\begin{aligned} \text{interp } c (v^\rightarrow) &= \text{just } (\delta (c, v)^\rightarrow) \\ \text{interp } (c_1 \oplus c_2) (\text{inj}_1 x^\rightarrow) &= \text{interp } c_1 (x^\rightarrow) \gg= \\ &\quad \lambda \{ (x'^\rightarrow) \rightarrow \text{just } (\text{inj}_1 x'^\rightarrow) ; \\ &\quad (x'^\leftarrow) \rightarrow \text{just } (\text{inj}_1 x'^\leftarrow) \} \\ \text{interp } (c_1 \oplus c_2) (\text{inj}_2 y^\rightarrow) &= \text{interp } c_2 (y^\rightarrow) \gg= \\ &\quad \lambda \{ (y'^\rightarrow) \rightarrow \text{just } (\text{inj}_2 y'^\rightarrow) ; \\ &\quad (y'^\leftarrow) \rightarrow \text{just } (\text{inj}_2 y'^\leftarrow) \} \\ \text{interp } (c_1 \otimes c_2) ((x, y)^\rightarrow) &= \text{interp } c_1 (x^\rightarrow) \gg= \\ &\quad \lambda \{ (x'^\leftarrow) \rightarrow \text{just } ((x', y)^\leftarrow) ; \\ &\quad (x'^\rightarrow) \rightarrow \text{interp } c_2 (y^\rightarrow) \gg= \\ &\quad \quad \lambda \{ (y'^\rightarrow) \rightarrow \text{just } ((x', y')^\rightarrow) ; \\ &\quad \quad (y'^\leftarrow) \rightarrow \text{just } ((x, y')^\leftarrow) \} \\ \text{interp } (\eta_* v) (\text{tt}^\rightarrow) &= \text{just } ((v, \odot)^\rightarrow) \\ \text{interp } (\varepsilon_* v) ((v', \odot)^\rightarrow) &\text{ with } v \stackrel{?}{=} v' \\ \dots \mid \text{yes } _ &= \text{just } (\text{tt}^\rightarrow) \\ \dots \mid \text{no } _ &= \text{nothing} \end{aligned}$$

η_+ and ε_+ turn the directions around:

$$\begin{aligned} \text{interp } \varepsilon_+ (\text{inj}_1 v^\rightarrow) &= \text{just } (\text{inj}_2 (-v)^\leftarrow) \\ \text{interp } \varepsilon_+ (\text{inj}_2 (-v)^\rightarrow) &= \text{just } (\text{inj}_1 v^\leftarrow) \\ \text{interp } \eta_+ (\text{inj}_1 v^\leftarrow) &= \text{just } (\text{inj}_2 (-v)^\rightarrow) \\ \text{interp } \eta_+ (\text{inj}_2 (-v)^\leftarrow) &= \text{just } (\text{inj}_1 v^\rightarrow) \end{aligned}$$

For the composition, we will use the handlers described in the interpreter of Π^- and the maybe monad:

$$\begin{aligned} \text{interp } (c_1 \circledast c_2) (v \rightarrow) &= \text{interp } c_1 (v \rightarrow) \gg= \\ &\lambda \{ (v' \leftarrow) \rightarrow \text{just } (v' \leftarrow) ; \\ &\quad (v' \rightarrow) \rightarrow \text{handle}\triangleright (c_1, v', c_2) \} \end{aligned}$$

The handlers work the same as described in Sec. 3.3, but the result will channel through the bind operator for the maybe monad:

$$\begin{aligned} \text{handle}\triangleright (c_1, b, c_2) &= \text{interp } c_2 (b \rightarrow) \gg= \\ &\lambda \{ (c \rightarrow) \rightarrow \text{just } (c \rightarrow) ; \\ &\quad (b' \leftarrow) \rightarrow \text{handle}\triangleleft (c_1, b', c_2) \} \end{aligned}$$

$$\begin{aligned} \text{handle}\triangleleft (c_1, b, c_2) &= \text{interp } c_1 (b \leftarrow) \gg= \\ &\lambda \{ (a \leftarrow) \rightarrow \text{just } (a \leftarrow) ; \\ &\quad (b' \rightarrow) \rightarrow \text{handle}\triangleright (c_1, b', c_2) \} \end{aligned}$$

The backward evaluation is similar:

$$\begin{aligned} \text{interp } \text{unite}_{+l} (v \leftarrow) &= \text{just } (\text{inj}_2 v \leftarrow) \\ \text{interp } \text{uniti}_{+l} (\text{inj}_2 v \leftarrow) &= \text{just } (v \leftarrow) \\ \text{interp } \text{swap}_{+} (\text{inj}_1 x \leftarrow) &= \text{just } (\text{inj}_2 x \leftarrow) \\ \text{interp } \text{swap}_{+} (\text{inj}_2 y \leftarrow) &= \text{just } (\text{inj}_1 y \leftarrow) \\ \text{interp } \text{assocl}_{+} (\text{inj}_1 (\text{inj}_1 x) \leftarrow) &= \text{just } (\text{inj}_1 x \leftarrow) \\ \text{interp } \text{assocl}_{+} (\text{inj}_1 (\text{inj}_2 y) \leftarrow) &= \text{just } (\text{inj}_2 (\text{inj}_1 y) \leftarrow) \\ \text{interp } \text{assocl}_{+} (\text{inj}_2 z \leftarrow) &= \text{just } (\text{inj}_2 (\text{inj}_2 z) \leftarrow) \\ \text{interp } \text{assocr}_{+} (\text{inj}_1 x \leftarrow) &= \text{just } (\text{inj}_1 (\text{inj}_1 x) \leftarrow) \\ \text{interp } \text{assocr}_{+} (\text{inj}_2 (\text{inj}_1 y) \leftarrow) &= \text{just } (\text{inj}_1 (\text{inj}_2 y) \leftarrow) \\ \text{interp } \text{assocr}_{+} (\text{inj}_2 (\text{inj}_2 z) \leftarrow) &= \text{just } (\text{inj}_2 z \leftarrow) \\ \text{interp } \text{unite}_{*l} (v \leftarrow) &= \text{just } ((\text{tt}, v) \leftarrow) \\ \text{interp } \text{uniti}_{*l} ((\text{tt}, v) \leftarrow) &= \text{just } (v \leftarrow) \\ \text{interp } \text{swap}_{*} ((x, y) \leftarrow) &= \text{just } ((y, x) \leftarrow) \end{aligned}$$

$\text{interp assocl}_* ((x, y), z) \leftarrow = \text{just } ((x, (y, z)) \leftarrow)$
 $\text{interp assocr}_* ((x, (y, z)) \leftarrow = \text{just } ((x, y), z) \leftarrow)$
 $\text{interp dist } (\text{inj}_1 x, z) \leftarrow = \text{just } ((\text{inj}_1 x, z) \leftarrow)$
 $\text{interp dist } (\text{inj}_2 y, z) \leftarrow = \text{just } ((\text{inj}_2 y, z) \leftarrow)$
 $\text{interp factor } ((\text{inj}_1 x, z) \leftarrow = \text{just } (\text{inj}_1 (x, z) \leftarrow)$
 $\text{interp factor } ((\text{inj}_2 y, z) \leftarrow = \text{just } (\text{inj}_2 (y, z) \leftarrow)$
 $\text{interp id}\leftrightarrow (v \leftarrow) = \text{just } (v \leftarrow)$
 $\text{interp } (c_1 \wp c_2) (v \leftarrow) = \text{interp } c_2 (v \leftarrow) \gg =$
 $\lambda \{ (v' \rightarrow) \rightarrow \text{just } (v' \rightarrow) ;$
 $(v' \leftarrow) \rightarrow \text{handle}\triangleleft (c_1, v', c_2) \}$
 $\text{interp } (c_1 \oplus c_2) (\text{inj}_1 x \leftarrow) = \text{interp } c_1 (x \leftarrow) \gg =$
 $\lambda \{ (x' \leftarrow) \rightarrow \text{just } (\text{inj}_1 x' \leftarrow) ;$
 $(x' \rightarrow) \rightarrow \text{just } (\text{inj}_1 x' \rightarrow) \}$
 $\text{interp } (c_1 \oplus c_2) (\text{inj}_2 y \leftarrow) = \text{interp } c_2 (y \leftarrow) \gg =$
 $\lambda \{ (y' \leftarrow) \rightarrow \text{just } (\text{inj}_2 y' \leftarrow) ;$
 $(y' \rightarrow) \rightarrow \text{just } (\text{inj}_2 y' \rightarrow) \}$
 $\text{interp } (c_1 \otimes c_2) ((x, y) \leftarrow) = \text{interp } c_2 (y \leftarrow) \gg =$
 $\lambda \{ (y' \rightarrow) \rightarrow \text{just } ((x, y') \rightarrow) ;$
 $(y' \leftarrow) \rightarrow \text{interp } c_1 (x \leftarrow) \gg =$
 $\lambda \{ (x' \leftarrow) \rightarrow \text{just } ((x', y') \leftarrow) ;$
 $(x' \rightarrow) \rightarrow \text{just } ((x', y) \rightarrow) \}$
 $\text{interp } (\varepsilon_* v) (\text{tt} \leftarrow) = \text{just } ((v, \odot) \leftarrow)$
 $\text{interp } (\eta_* v) ((v', \odot) \leftarrow) \text{ with } v \stackrel{?}{=} v'$
 $\dots \mid \text{yes } _ = \text{just } (\text{tt} \leftarrow)$
 $\dots \mid \text{no } _ = \text{nothing}$

The interpreter implements the machine semantics described in Sec. 5.2.

Theorem 5.10. $\mathcal{U}_{\text{Agda}}$ For all c and V , either:

- $\text{interp}^{\mathbb{Q}}(c)(V) = \text{just } \text{eval}^{\mathbb{Q}}(c)(V)$ or
- $\text{interp}^{\mathbb{Q}}(c)(V) = \text{nothing}$ and $\text{eval}^{\mathbb{Q}}(c)(V) = \text{error}$.

Proof. By induction on c :

- c is a base combinator or $\text{id} \leftrightarrow$: By analyzing all possible cases.
- $c = \varepsilon_+, \eta_+, \eta_*^{v:t}$ and $\varepsilon_*^{v:t}$: By analyzing all possible cases.
- $c = c_1 \otimes c_2$: By analyzing all possible cases and the inductive hypothesis.
- $c = c_1 \oplus c_2$: By analyzing all possible cases and the inductive hypothesis.
- $c = c_1 \circ c_2$ where $c_1 : A \leftrightarrow B$ and $c_2 : B \leftrightarrow C$: There are two cases:
 - $V = v^\rightarrow$: There are three cases:
 - * $\text{eval}^{\mathbb{Q}}(c_1)(v^\rightarrow) = w^\leftarrow$: By the inductive hypothesis $\text{interp}^{\mathbb{Q}}(c_1)(v^\rightarrow) \Downarrow w^\leftarrow$.
Hence $\text{interp}^{\mathbb{Q}}(c_1 \circ c_2)(v^\rightarrow) = \text{just } w^\leftarrow = \text{just } \text{eval}^{\mathbb{Q}}(c_1 \circ c_2)(v^\rightarrow)$.
 - * $\text{eval}^{\mathbb{Q}}(c_1)(v^\rightarrow) = \text{error}$: By the inductive hypothesis $\text{interp}^{\mathbb{Q}}(c_1)(v^\rightarrow) = \text{nothing}$.
Hence $\text{interp}^{\mathbb{Q}}(c_1 \circ c_2)(v^\rightarrow) = \text{nothing}$ and $\text{eval}^{\mathbb{Q}}(c_1 \circ c_2)(v^\rightarrow) = \text{error}$.
 - * $\text{eval}^{\mathbb{Q}}(c_1)(v^\rightarrow) = w^\rightarrow$: there are three possible case of $\text{eval}^-(c_1 \circ c_2)(v^\rightarrow)$. If $\text{eval}^{\mathbb{Q}}(c_1 \circ c_2)(v^\rightarrow) = u^\rightarrow$, an additional lemma is needed for this case.

Lemma 5.11. $\mathcal{U}\text{Agda}$ For any $n \in \mathbb{N}$ and $b : B$, the following hold:

1. If $[c_1 \mid b \mid (\square \circ c_2) \bullet \square]_{\triangleright} \mapsto^n [c_1 \circ c_2 \mid u \mid \square]_{\triangleright}$ then $\text{handle}^{\triangleright}(c_1, b, c_2) = \text{just } u^\rightarrow$.
2. If $\langle c_2 \mid b \mid (c_1 \circ \square) \bullet \square \rangle_{\triangleleft} \mapsto^n [c_1 \circ c_2 \mid u \mid \square]_{\triangleright}$ then $\text{handle}^{\triangleleft}(c_1, b, c_2) = \text{just } u^\rightarrow$.

Proof. By induction on n :

- $n = 0$: No such execution exists, so it is vacuously true.
- $n > 0$:
 1. Assume $[c_1 \mid b \mid (\square \circ c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \circ c_2 \mid u \mid \square]_{\triangleright}$ and its length is n .
There are two possible outcomes of $\text{eval}^{\mathbb{Q}}(c_2)(b^\rightarrow)$, if $\text{eval}^{\mathbb{Q}}(c_2)(v^\rightarrow) = x^\rightarrow$ then

$$[c_1 \mid b \mid (\square \circ c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \circ c_2 \mid x \mid \square]_{\triangleright}$$

Since $\Pi^{\mathbb{Q}}$ -machine is deterministic so $x = u$ and $\text{interp}^{\mathbb{Q}}(c_2)(v^\rightarrow) = \text{just } u^\rightarrow$. Hence $\text{handle}^{\triangleright}(c_1, v^\rightarrow, c_2) = u^\rightarrow$.

If $\text{eval}^-(c_2)(v^\rightarrow) = x^\leftarrow$ then

$$[c_1 \mid b \mid (\square \circ c_2) \bullet \square]_{\triangleright} \mapsto^* \langle c_2 \mid x \mid (c_1 \circ \square) \bullet \square \rangle_{\triangleleft} \mapsto^* [c_1 \circ c_2 \mid u \mid \square]_{\triangleright}$$

The length of $\langle c_2 \mid x \mid (c_1 \mathbin{\text{;}} \square) \bullet \square \rangle_{\triangleleft} \mapsto^* [c_1 \mathbin{\text{;}} c_2 \mid u \mid \square]_{\triangleright}$ is less than n , so by inductive hypothesis $\text{handle}_{\triangleleft}(c_1, b^{\leftarrow}, x) = \text{just } u^{\rightarrow}$. Therefore $\text{handle}_{\triangleright}(c_1, v^{\rightarrow}, c_2) = \text{just } u^{\rightarrow}$.

2. Similar.

□

Since $\text{eval}^{\mathbb{Q}}(c_1)(v^{\rightarrow}) = w^{\rightarrow}$ so $\text{interp}^{\mathbb{Q}}(c_1)(v^{\rightarrow}) = \text{just } w^{\rightarrow}$ and

$$\langle c_1 \mathbin{\text{;}} c_2 \mid v \mid \square \rangle_{\triangleright} \mapsto^* [c_1 \mid w \mid (\square \mathbin{\text{;}} c_2) \bullet \square]_{\triangleright} \mapsto^* [c_1 \mathbin{\text{;}} c_2 \mid u \mid \square]_{\triangleright}$$

By Lemma 5.11, $\text{handle}_{\triangleright}(c_1, w, c_2) = \text{just } u^{\rightarrow}$. Therefore $\text{interp}^-(c_1 \mathbin{\text{;}} c_2)(v^{\rightarrow}) = \text{just } u^{\rightarrow}$.

The case of $\text{eval}^{\mathbb{Q}}(c_1 \mathbin{\text{;}} c_2)(v^{\rightarrow}) = u^{\leftarrow}$ and $\text{eval}^{\mathbb{Q}}(c_1 \mathbin{\text{;}} c_2)(v^{\rightarrow}) = \text{error}$ are similar.
 – $V = v^{\leftarrow}$: Similar.

□

5.4 EXAMPLES

This section will introduce some example $\Pi^{\mathbb{Q}}$ programs, which demonstrate the expressiveness of fractional and negative types. In the end, we will give an implementation of a SAT solver which utilizes both additive and multiplicative traces.

$\Pi^{\mathbb{Q}}$ allow us to give computational interpretations to equations that are valid in the field of rational numbers:

$$\begin{aligned} \text{inv-} & : \{A : \mathbb{U}\} \rightarrow A \leftrightarrow - (- A) \\ \text{inv-} & = \text{uniti}_{+r} \mathbin{\text{;}} (\text{id} \leftrightarrow \oplus \eta_+) \mathbin{\text{;}} \text{assocl}_+ \mathbin{\text{;}} (\varepsilon_+ \oplus \text{id} \leftrightarrow) \mathbin{\text{;}} \text{unite}_{+l} \\ \text{dist-} & : \{A \ B : \mathbb{U}\} \rightarrow - (A +_u B) \leftrightarrow - A +_u - B \\ \text{dist-} & = \text{uniti}_{+l} \mathbin{\text{;}} (\eta_+ \oplus \text{id} \leftrightarrow) \mathbin{\text{;}} \text{uniti}_{+l} \mathbin{\text{;}} (\eta_+ \oplus \text{id} \leftrightarrow) \mathbin{\text{;}} \text{assocl}_+ \\ & \mathbin{\text{;}} ([A+B]+[C+D]=[A+C]+[B+D] \oplus \text{id} \leftrightarrow) \\ & \mathbin{\text{;}} (\text{swap}_+ \oplus \text{id} \leftrightarrow) \mathbin{\text{;}} \text{assocr}_+ \mathbin{\text{;}} (\text{id} \leftrightarrow \oplus \varepsilon_+) \mathbin{\text{;}} \text{unite}_{+r} \\ \text{neg-} & : \{A \ B : \mathbb{U}\} \rightarrow (A \leftrightarrow B) \rightarrow (- A \leftrightarrow - B) \\ \text{neg- } c & = \text{uniti}_{+r} \mathbin{\text{;}} (\text{id} \leftrightarrow \oplus \eta_+) \end{aligned}$$

$$\begin{aligned} & \textcircled{\circ} (\text{id} \leftrightarrow \oplus ! c \oplus \text{id} \leftrightarrow) \textcircled{\circ} \text{assocl}_+ \\ & \textcircled{\circ} ((\text{swap}_+ \textcircled{\circ} \varepsilon_+) \oplus \text{id} \leftrightarrow) \textcircled{\circ} \text{unite}_{+l} \end{aligned}$$

$$\text{inv}/ : \{A : \mathbb{U}\} \{v : \llbracket A \rrbracket\} \rightarrow A \leftrightarrow (\mathbb{1}/_ \{ \mathbb{1}/ v \} \circ)$$

$$\text{inv}/ = \text{uniti}_{*r} \textcircled{\circ} (\text{id} \leftrightarrow \otimes \eta_* _) \textcircled{\circ} \text{assocl}_* \textcircled{\circ} (\varepsilon_* _ \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{unite}_{*l}$$

$$\text{dist}/ : \{A B : \mathbb{U}\} \{a : \llbracket A \rrbracket\} \{b : \llbracket B \rrbracket\} \rightarrow \mathbb{1}/ (a , b) \leftrightarrow \mathbb{1}/ a \times_u \mathbb{1}/ b$$

$$\begin{aligned} \text{dist}/ \{A\} \{B\} \{a\} \{b\} &= \text{uniti}_{*l} \textcircled{\circ} (\eta_* b \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{uniti}_{*l} \textcircled{\circ} (\eta_* a \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{assocl}_* \\ &\textcircled{\circ} ([A \times B] \times [C \times D] = [A \times C] \times [B \times D]) \otimes \text{id} \leftrightarrow \\ &\textcircled{\circ} (\text{swap}_* \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{assocr}_* \textcircled{\circ} (\text{id} \leftrightarrow \otimes \varepsilon_* (a , b)) \textcircled{\circ} \text{unite}_{*r} \end{aligned}$$

$$\text{neg}/ : \{A B : \mathbb{U}\} \{a : \llbracket A \rrbracket\} \{b : \llbracket B \rrbracket\} \rightarrow (A \leftrightarrow B) \rightarrow (\mathbb{1}/ a \leftrightarrow \mathbb{1}/ b)$$

$$\begin{aligned} \text{neg}/ \{A\} \{B\} \{a\} \{b\} c &= \text{uniti}_{*r} \textcircled{\circ} (\text{id} \leftrightarrow \otimes \eta_* _) \\ &\textcircled{\circ} (\text{id} \leftrightarrow \otimes ! c \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{assocl}_* \\ &\textcircled{\circ} ((\text{swap}_* \textcircled{\circ} \varepsilon_* _) \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{unite}_{*l} \end{aligned}$$

$$\text{dist}\times- : \{A B : \mathbb{U}\} \rightarrow (- A) \times_u B \leftrightarrow - (A \times_u B)$$

$$\begin{aligned} \text{dist}\times- &= \text{uniti}_{+r} \textcircled{\circ} (\text{id} \leftrightarrow \oplus \eta_+) \textcircled{\circ} \text{assocl}_+ \\ &\textcircled{\circ} ((\text{factor} \textcircled{\circ} ((\text{swap}_+ \textcircled{\circ} \varepsilon_+) \otimes \text{id} \leftrightarrow)) \textcircled{\circ} \text{absorbr}) \oplus \text{id} \leftrightarrow \\ &\textcircled{\circ} \text{unite}_{+l} \end{aligned}$$

$$\text{neg}\times : \{A B : \mathbb{U}\} \rightarrow A \times_u B \leftrightarrow (- A) \times_u (- B)$$

$$\text{neg}\times = \text{inv-} \textcircled{\circ} \text{neg-} \textcircled{\circ} (! \text{dist}\times- \textcircled{\circ} \text{swap}_*) \textcircled{\circ} ! \text{dist}\times- \textcircled{\circ} \text{swap}_*$$

$$\text{fracDist} : \forall \{A B\} \{a : \llbracket A \rrbracket\} \{b : \llbracket B \rrbracket\} \rightarrow \mathbb{1}/ a \times_u \mathbb{1}/ b \leftrightarrow \mathbb{1}/ (a , b)$$

$$\begin{aligned} \text{fracDist} &= \text{uniti}_{*l} \textcircled{\circ} ((\eta_* _ \textcircled{\circ} \text{swap}_*) \otimes \text{id} \leftrightarrow) \textcircled{\circ} \text{assocr}_* \\ &\textcircled{\circ} (\text{id} \leftrightarrow \otimes ([A \times B] \times [C \times D] = [A \times C] \times [B \times D]) \textcircled{\circ} (\varepsilon_* _ \otimes \varepsilon_* _) \textcircled{\circ} \text{unite}_{*l}) \textcircled{\circ} \text{unite}_{*r} \end{aligned}$$

$$\text{mulFrac} : \forall \{A B C D\} \{b : \llbracket B \rrbracket\} \{d : \llbracket D \rrbracket\}$$

$$\rightarrow (A \times_u \mathbb{1}/ b) \times_u (C \times_u \mathbb{1}/ d) \leftrightarrow (A \times_u C) \times_u (\mathbb{1}/ (b , d))$$

$$\text{mulFrac} = [A \times B] \times [C \times D] = [A \times C] \times [B \times D] \textcircled{\circ} (\text{id} \leftrightarrow \otimes \text{fracDist})$$

$$\text{addFracCom} : \forall \{A B C\} \{v : \llbracket C \rrbracket\}$$

$$\rightarrow (A \times_u \mathbb{1}/ v) +_u (B \times_u \mathbb{1}/ v) \leftrightarrow (A +_u B) \times_u (\mathbb{1}/ v)$$

`addFracCom = factor`

```

addFrac : ∀ {A B C D} → (v : [ C ]) → (w : [ D ])
  → (A ×u 1/_ {C} v) +u (B ×u 1/_ {D} w) ↔
    ((A ×u D) +u (C ×u B)) ×u (1/_ {C ×u D} (v , w))
addFrac v w = ((uniti*r ; (id↔ ⊗ η* w)) ⊕ (uniti*l ; (η* v ⊗ id↔)))
  ; [A×B]×[C×D]=[A×C]×[B×D] ⊕ [A×B]×[C×D]=[A×C]×[B×D]
  ; ((id↔ ⊗ fracDist) ⊕ (id↔ ⊗ fracDist)) ; factor

```

Using the constructs from Sec. 3.5 and 4.5, a SAT solver can be implemented in Π^Q . A SAT solver's job is to check the satisfiability of a given function $F : \mathbb{B}^n \rightarrow \mathbb{B}$, i.e., it succeeds if $\exists \bar{b} : \mathbb{B}^n.F(\bar{b}) = \mathbb{T}$. Since Π^Q only contains reversible programs, the first step would be to construct a reversible version of F using Toffoli's construction [42]. Using Toffoli's construction we can obtain $F^r : \mathbb{B}^{1+n} \rightarrow \mathbb{B}^{1+n}$ which satisfies $\forall \bar{b} : \mathbb{B}^n.F^r(\mathbb{F}, \bar{b}) = (F(\bar{b}), _)$. Now, our goal is to construct a circuit to find out whether $\exists \bar{b} : \mathbb{B}^n.F^r(\mathbb{F}, \bar{b}) = (\mathbb{T}, \dots)$ or not. To simplify the circuit, we will solve an equivalent problem: $\exists \bar{b} : \mathbb{B}^n. \sim F^r(\mathbb{F}, \bar{b}) = (\mathbb{F}, \dots)$ where $\sim F^r = F^r ; (\text{NOT} \otimes \text{id} \leftrightarrow)$ (reversible version of $\sim F = \text{NOT} \circ F$). It is obvious that the invalidity of $\sim F$ implies the satisfiability of F :

$$\exists \bar{b} : \mathbb{B}^{1+n}.F^r(\mathbb{F}, \bar{b}) = (\mathbb{T}, \dots) \text{ iff } \exists \bar{b} : \mathbb{B}^{1+n}. \sim F^r(\mathbb{F}, \bar{b}) = (\mathbb{F}, \dots)$$

One approach to check the invalidity of $\sim F$ is to loop through all the possible inputs:

```
for ( $\bar{b} = \mathbb{F}^{1+n}$ ;  $\sim F^r(\bar{b}) = (\mathbb{T}, \_)$ ;  $\bar{b}++$ );
```

Since $\sim F^r$ is reversible, there will always exist $\bar{b} : \mathbb{B}^{1+n}$ such that $F^r(\bar{b}) = (\mathbb{F}, \dots)$. And since we tried all the possible values from the smallest to the largest, we will always enumerate all $(\mathbb{F}, \bar{b}') : \mathbb{B}^{1+n}$ before $(\mathbb{T}, \bar{b}') : \mathbb{B}^{1+n}$. Hence, we can simply check the first bit of \bar{b} after the loop to find out whether $\exists \bar{b}' : \mathbb{B}^n. \sim F^r(\mathbb{F}, \bar{b}') = (\mathbb{F}, \dots)$ or not:

$$\bar{b} = (\mathbb{F}, \dots) \text{ iff } \exists \bar{b}' : \mathbb{B}^n. \sim F^r(\mathbb{F}, \bar{b}') = (\mathbb{F}, \dots)$$

The **LOOP** from Sec. 3.5 does exactly what we want, when the output is \mathbb{F}^{2+n} , $\text{LOOP}_{\sim F^r} : \mathbb{B}^{2+n} \leftrightarrow \mathbb{B}^{2+n}$ will compute:

```

for ( $\bar{b} = \mathbb{F}^{1+n}; \sim F^r(\bar{b}) = (\mathbb{T}, \_); \bar{b}++$ );
return ( $\mathbb{F}, \bar{b}$ );

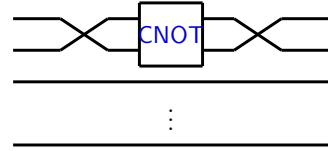
```

To obtain the desired results from $\text{LOOP}_{\sim F^r}$ requires the input to be \mathbb{F}^{2+n} which can be provided via using the multiplicative trace. When $\text{LOOP}_{\sim F^r}$ finishes, the output will be (\mathbb{F}, \bar{b}) where the first bit of \bar{b} contains the information about the invalidity of $\sim F$ and we would like to keep it, so we will use the **COPY** to do that. **COPY** uses the property of **CNOT**: $\text{CNOT}(x, \mathbb{F}) = (x, x)$. Utilizing this property we can implement **COPY** such that $\text{COPY}(\mathbb{F}, b_1, \dots) = (b_1, b_1, \dots)$:

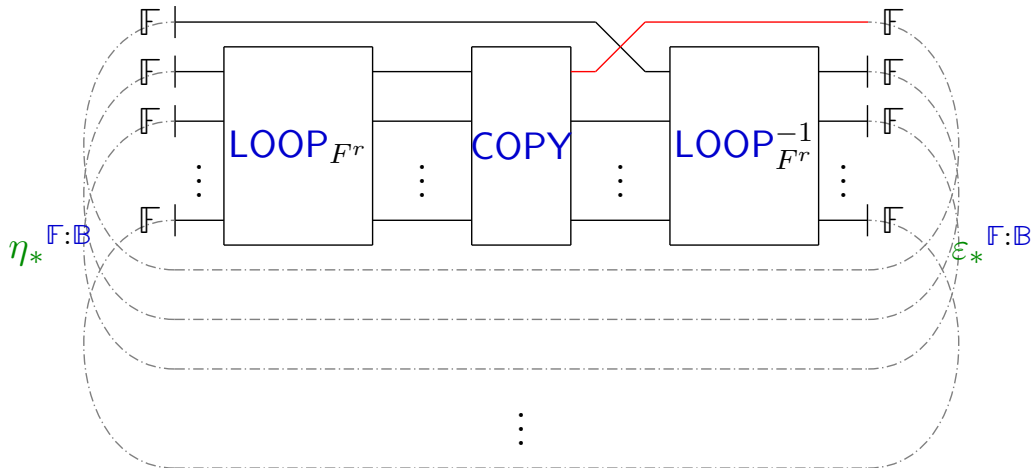
$\text{COPY} \{1\} = \text{swap}_* \ ; \ \text{CNOT} \ ; \ \text{swap}_*$

$\text{COPY} \{\text{suc}(\text{suc } n)\} =$

$\text{assocl}_* \ ; \ (\text{COPY} \{1\} \otimes \text{id} \leftrightarrow) \ ; \ \text{assocr}_*$



And after copying the result, we will need to uncompute \bar{b} so that the allocated wire can be garbage collected properly. This is a simple job because Π^Q is a reversible programming language so we can simply use the inverse of $\text{LOOP}_{\sim F^r}$ denoted by $\text{LOOP}_{\sim F^r}^{-1}$. However, there is one problem. The output of $\text{LOOP}_{\sim F^r}$ is (\mathbb{F}, \bar{b}) , but after copying it becomes (b_1, \bar{b}) . To uncompute \bar{b} back to \mathbb{F}^{1+n} we will need another \mathbb{F} wire, so the multiplicative trace would need to allocate $3 + n$ \mathbb{F} wires. The whole construction is given as follows:



$$\text{SAT} : \forall \{n\} \rightarrow \mathbb{B}^\wedge(1+n) \leftrightarrow \mathbb{B}^\wedge(1+n) \rightarrow \mathbb{1} \leftrightarrow \mathbb{1}$$

$$\text{SAT } \{n\} F^r = \text{trace}_* (\mathbb{F}^\wedge(3+n))$$

$$\begin{aligned} & (\text{id} \leftrightarrow \otimes ((\text{id} \leftrightarrow \otimes (\text{LOOP } (\sim F^r) \text{ ; } (\text{id} \leftrightarrow \otimes \text{SPLIT } 1 \ n))) \\ & \text{ ; } (\text{id} \leftrightarrow \otimes (\text{assocl}_* \text{ ; } (\text{COPY} \otimes \text{id} \leftrightarrow) \text{ ; } \text{assocr}_*)) \\ & \text{ ; } \text{assocl}_* \text{ ; } (\text{swap}_* \otimes \text{id} \leftrightarrow) \text{ ; } \text{assocr}_* \\ & \text{ ; } (\text{id} \leftrightarrow \otimes ((\text{id} \leftrightarrow \otimes \text{MERGE } 1 \ n) \text{ ; } \text{LOOP}^{-1} (\sim F^r)))))) \end{aligned}$$

Note that the red wire contains the answer to our SAT instance, if it is \mathbb{F} then F is satisfiable. So if F is satisfiable the garbage collection will succeed, and it will throw an exception if F is unsatisfiable (We could also leave that answer wire uncollected if we want to use it in the following computation, at the expense of keeping its GC token around).

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this work, we discussed reversible effects inspired by compact closed categories. Starting from a programming language Π which has syntax from bimonoidal groupoid, we extend it to compact closed categories by proposing that negative and fractional types express backtracking in “time” and “space.” As a result, we give a computational interpretation of compact closed categories. To demonstrate that negative and fractional types have applications in programming, various examples have been discussed. The whole work is formalized in Agda. The following are some possible future directions based on this work:

- **Extension of Π :** The next important natural extension of Π after this work is adding recursive types. It will greatly increase the expressiveness of Π . It has been shown in this work that with the presence of effects, extended Π still guarantees termination. However, it is nontrivial to figure out the right combinators for recursive types that preserve termination. Even without the effects introduced in this work, it is already challenging to come up with a language that captures all the computable bijections between recursive types, which has a foundational importance in reversible computation and recursion theory.
- **Quantum Computing:** Compact closed categories have been used in modeling quantum computing [1]. Now that we have operational semantics for compact closed categories. It is possible to relate it to quantum computing through compact closed categories. It will be interesting to fully explore how these two concepts relate to each other. The connection between them might bring deeper insights into the understanding of quantum theory.

On the practical side, the reversible extensions proposed in this work can be applied directly to quantum programming languages to provide high-level abstractions such as backtracking, which will greatly enrich the expressiveness.

- Meadows: The axioms of meadows [9] ($x \cdot (x \cdot \frac{1}{x}) = x$ and $\frac{1}{\frac{1}{x}} = x$) provide another approach to handle the division by zero problem in fractional types. Combining with the idea from ancilla managements in [14], we might be able to obtain a computational interpretation for the categorification of meadows.
- Concurrent Programming: *Future* in concurrent programming is a perfect fit for fractional types. As a *Future* carries data of type t , its putter has type $\frac{1}{t}$ and its getter has type t . If the language only has product types, this leads to another operational semantics for compact closed categories. However, if we add sum types to the language, the computation might get stuck. It would be interesting to find out how to incorporate sum types into this operational semantics. This incorporation will result in a concurrent computational interpretation of compact closed categories and a type-theoretic way to guarantee safe usages of *Future*.
- Session types: Since intuitionistic linear logic can be interpreted as session types [10], it is possible to connect negative types with session types. I conjecture that they should be able to interpret each other: the session receives a value of type t is corresponding to the negative type $-t$, and the session sends a value of type t is corresponding to the positive type t .

BIBLIOGRAPHY

- [1] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 415–425, 2004.
- [2] Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441 – 464, 2005.
- [3] Samson Abramsky, Simon Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, page 35–113, Berlin, Heidelberg, 1996. Springer-Verlag.
- [4] Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Comp. Sci.*, 12(5):625–665, October 2002.
- [5] Samson Abramsky and Radha Jagadeesan. New foundations for the geometry of interaction. *Inf. Comput.*, 111:53–119, May 1994.
- [6] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532, November 1973.
- [7] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.
- [8] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, pages 42–122, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

- [9] J.A. Bergstra, Y. Hirshfeld, and J.V. Tucker. Meadows and the equational specification of division. *Theoretical Computer Science*, 410(12):1261 – 1271, 2009.
- [10] Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.
- [11] Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. From reversible programs to univalent universes and back. *Electronic Notes in Theoretical Computer Science*, 336:5 – 25, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).
- [12] Jacques Carette and Amr Sabry. Computing with semirings and weak rig groupoids. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, page 123–148, Berlin, Heidelberg, 2016. Springer-Verlag.
- [13] Siu Man Chan. *Pebble Games and Complexity*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2013.
- [14] Chao-Hong Chen, Vikraman Choudhury, Jacques Carette, and Amr Sabry. Fractional types. In Ivan Lanese and Mariusz Rawski, editors, *Reversible Computation*, pages 169–186. Springer International Publishing, 2020.
- [15] Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: Reversible programming with negative and fractional types. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.
- [16] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
- [17] Robin Cockett and Chris Heunen. Compact inverse categories, 2019.
- [18] Cole Comfort, Antonin Delpeuch, and Jules Hedges. Sheet diagrams for bimonoidal categories, 2020.

- [19] Jadav Chandra Das and Debashis De. Novel low power reversible binary incremter design using quantum-dot cellular automata. *Microprocessors and Microsystems*, 42:10 – 23, 2016.
- [20] Jadav Chandra Das and Debashis De. Quantum-dot cellular automata based reversible low power parity generator and parity checker design for nanocommunication. *Frontiers of Information Technology & Electronic Engineering*, 17(3):224–236, 2016.
- [21] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 77–88, New York, NY, USA, 2004. Association for Computing Machinery.
- [22] Marcelo Fiore. An axiomatics and a combinatorial model of creation/annihilation operators. [arXiv:1506.06402\[math.CT\]](https://arxiv.org/abs/1506.06402), 2015.
- [23] Marcelo Fiore, Roberto [Di Cosmo], and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1):35 – 50, 2006.
- [24] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.
- [25] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 333–342, New York, NY, USA, 2013. Association for Computing Machinery.
- [26] Furio Honsell and Marina Lenisa. “wave-style” geometry of interaction models in rel are graph-like lambda-models. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 242–258, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

- [27] Roshan P. James and Amr Sabry. Information effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 73–84, New York, NY, USA, 2012. Association for Computing Machinery.
- [28] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.
- [29] J. Kastl. Inverse categories. *Studien zur Algebra und ihre Anwendungen*, 7:51–60, 1979.
- [30] G. M. Kelly. Coherence theorems for lax algebras and for distributive laws. In Gregory M. Kelly, editor, *Category Seminar*, pages 281–375, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.
- [31] G Maxwell Kelly. Many-variable functorial calculus. i. In *Coherence in categories*, pages 66–105. Springer, 1972.
- [32] G.M. Kelly and M.L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193 – 213, 1980.
- [33] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.
- [34] Miguel L Laplaza. Coherence for distributivity. In *Coherence in categories*, pages 29–65. Springer, 1972.
- [35] Miguel L. Laplaza. Coherence for distributivity. In G. M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane, editors, *Coherence in Categories*, pages 29–65, Berlin, Heidelberg, 1972. Springer Berlin Heidelberg.
- [36] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.
- [37] Saunders MacLane. Natural associativity and commutativity. *Rice Institute Pamphlet-Rice University Studies*, 49(4), 1963.
- [38] Amr Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, January 1998.

- [39] Zachary Sparks and Amr Sabry. Superstructural reversible logic. In *3rd International Workshop on Linearity*. Citeseer, 2014.
- [40] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language rfun. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [41] Michael Kirkedal Thomsen, Robin Kaarsgaard, and Mathias Soeken. Ricercar: A language for describing and rewriting reversible circuits with ancillae and its permutation semantics. In Jean Krivine and Jean-Bernard Stefani, editors, *Reversible Computation*, pages 200–215, Cham, 2015. Springer International Publishing.
- [42] Tommaso Toffoli. Reversible computing. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 632–644, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.
- [43] Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.
- [44] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In *Proceedings of the Third International Conference on Reversible Computation, RC'11*, page 14–29, Berlin, Heidelberg, 2011. Springer-Verlag.
- [45] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '07*, page 144–153, New York, NY, USA, 2007. Association for Computing Machinery.

CHAO-HONG CHEN

chen464@iu.edu

<https://dreamlinuxer.github.io/>

EDUCATION

- Doctor of Philosophy**, Computer Science
Indiana University, US *2015 - 2021*
- Master of Science**, Computer Science and Engineering
National Chiao Tung University, Taiwan *2007 - 2009*
- Bachelor of Science**, Computer Science
National Chiao Tung University, Taiwan *2003 - 2007*

WORKING EXPERIENCE

- Indiana University** 2015 - 2021
Associate Instructor *IN, US*
- Computing Theory, 2015 Fall
 - Elem Artificial Intelligence, 2017 Fall
 - Elem Artificial Intelligence, 2018 Spring
 - Fundamentals of Computing Theory, 2018 Fall
 - Fundamentals of Computing Theory, 2019 Spring
 - Applied Algorithms, 2019 Fall
 - Machine Learning, 2020 Spring
 - Applied Algorithms, 2020 Fall
 - Applied Algorithms, 2021 Spring
- Incentia Design Systems, Inc.** 2011 - 2014
Software Engineer *Hsinchu, Taiwan*
- Performance improvement
 - Multi-core programming
 - Software maintenance
 - C/C++ programming
- National Chiao Tung University** August 2010 - June 2011
Research Assistant *Hsinchu, Taiwan*
- Studies on the Application of General Optimization Frameworks to the Optimization Problems in Wireless Networking Technologies
- National Chiao Tung University** September 2007 - June 2011, September 2014 - June 2015
Teaching Assistant *Hsinchu, Taiwan*
- Formal Languages and Theory of Computation, 2014 Fall
 - Introduction to Artificial Intelligence, 2011 Spring
 - Introduction to Formal Language, 2010 Fall
 - Artificial Intelligence, 2010 Spring
 - Introduction to Formal Language, 2009 Fall

· Mathematical Logic, 2007 Fall

RESEARCH INTERESTS

Programming Language, Dependently-typed programming, Logic, Type Theory, Reversible Computation, Evolutionary Computation.

PROGRAMMING LANGUAGES

Agda, C/C++, Haskell, Coq, Ocaml, Python, Racket, SML.

AWARD

2007 ACM SIGEVO GECCO Student Travel Awards
National College Programming Contest 2007 Third Place
2006 ACM SIGEVO GECCO Student Travel Awards
National College Programming Contest 2006 Honorable Mention
National College Programming Contest 2005 Third Place
The 2005 ACM Asia Programming Contest Taipei Site Ninth Place
National College Programming Contest 2004 Honorable Mention
The 2003 ACM Asia Programming Contest Kaohsiung Site Seventh Place

SCHOLARSHIP

Entrance Scholarship of Institute of Computer Science and Engineering 2007, National Chiao Tung University
College Student Research Participation Fellowship, July 2006 - February 2007, National Science Council (Project: Discretization for Probabilistic Model Building Genetic Algorithms in Solving Optimization Problem in Continuous Domain)

PUBLICATION

- [1] **Chao-Hong Chen** and Amr Sabry. A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types. 2021 Symposium on Principles of Programming Languages (POPL).
- [2] Fang-Yi Lo, **Chao-Hong Chen** and Ying-ping Chen. Shrinking Counterexamples in Property-Based Testing with Genetic Algorithms. 2020 IEEE Congress on Evolutionary Computation (CEC).
- [3] **Chao-Hong Chen**, Vikraman Choudhury, Jacques Carette, Amr Sabry. Fractional Types: Expressive and Safe Space Management for Ancilla Bits. In Proceedings of the 12th international conference on Reversible Computation (RC'20).
- [4] Fang-Yi Lo and **Chao-Hong Chen** and Ying-ping Chen. Genetic Algorithms As Shrinkers in Property-based Testing. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19). ACM, New York, NY, USA, 291-292.
- [5] Jacques Carette, **Chao-Hong Chen**, Vikraman Choudhury and Amr Sabry. From Reversible Programs to Univalent Universes and Back. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). 2018.
- [6] **Chao-Hong Chen**, Vikraman Choudhury, and Ryan R. Newton. Adaptive lock-free data structures in Haskell: a general method for concurrent implementation swapping. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017). ACM, New York, NY, USA, 197-211.

- [7] Li-An Yang, Jui-Bin Liu, **Chao-Hong Chen** and Ying-ping Chen. Automatically Proving Mathematical Theorems with Evolutionary Algorithms and Proof Assistants. 2016 IEEE Congress on Evolutionary Computation (CEC), Vancouver, BC, 2016, pp. 4421-4428.
- [8] **Chao-Hong Chen** and Ying-Ping Chen. Quality analysis of discretization methods for estimation of distribution algorithms. *IEICE Transactions*, 97-D(5):1312–1323, 2014.
- [9] **Chao-Hong Chen** and Ying-Ping Chen. Convergence time analysis of particle swarm optimization based on particle interaction. *Adv. Artificial Intelligence*, 2011, 2011.
- [10] Ying-Ping Chen and **Chao-Hong Chen**. Enabling the extended compact genetic algorithm for real-parameter optimization by using adaptive discretization. *Evol. Comput.*, 18(2):199–228, June 2010.
- [11] **Chao-Hong Chen** and Ying-ping Chen. Real-coded ecga for economic dispatch. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1920–1927, New York, NY, USA, 2007. ACM.
- [12] **Chao-Hong Chen**, Wei-Nan Liu, and Ying-Ping Chen. Adaptive discretization for probabilistic model building genetic algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1103–1110, New York, NY, USA, 2006. ACM.