Agnideepa Sinha

Verified Compilation to A Reversible Language

Computer Science Tripos – Part II

Churchill College

May 2021

Declaration of Originality

I, Agnideepa Sinha of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed Agnideepa Sinha

Date May 14, 2021

Proforma

Name:	**2433E**
Project Title:	**Verified Compilation to A Reversible Language**
Examination:	**Computer Science Tripos – Part II, May 2021**
Word Count:	**11,987**[1]
Code line Count:	**1350**[2]
Project Originator:	Prof. Tim Griffin
Supervisor:	Prof. Tim Griffin

Original Aims of the Project

The original aim of this project was to define two languages: a conventional high-level irreversible language called LET and a low-level reversible language called Π, and then to write a formally verified compiler (in Agda) for translation from the former to the latter. The languages and translations are based on [10] by James and Sabry. The major contribution of this project was to be a formal verification of their translation for a strongly normalising (terminating) subset of the languages.

Work Completed

There are two important components in this project: a low-level functional reversible language called Π (the extended version is called Π°) and translation to it from a conventional high-level irreversible language LET. I have formalised these in Agda, as well as formally verified the translation and the reversibility of Π (and Π°). I have met all my core goals, along with significant extensions that allowed Π° to be Turing-complete as I show in my evaluation. To the best of my knowledge, a compilation of this form has never previously been formalised in Agda or any other proof assistant.

Special Difficulties

None.

[1] This word count was computed by `https://app.uio.no/ifi/texcount`.

[2] This line count includes lagda commands used to compile Agda code to LaTeX.

Contents

Chapter 1

Introduction

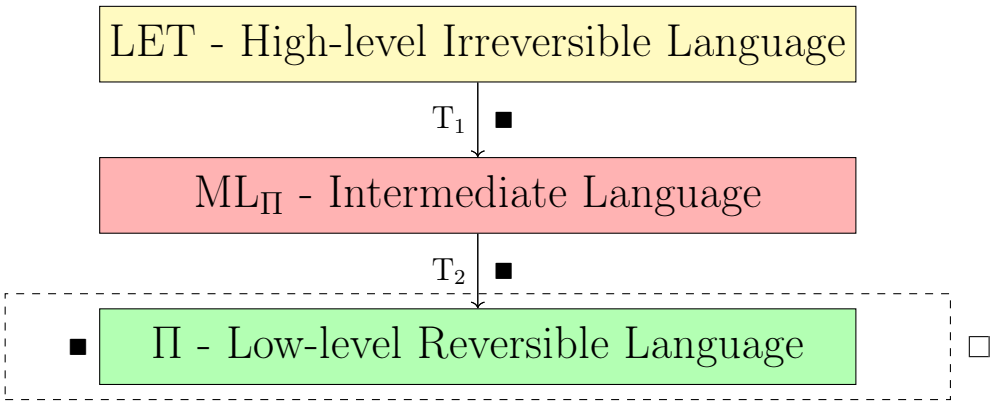


Figure 1.1: An overview of this project

The above languages and translations are based on the paper 'Information Effects' [10] by James and Sabry. Their scheme for compilation introduces code that explicitly models the heap and garbage and results in reversible compiled code even when the source program is irreversible. With some extensions and modifications, I have formalised and verified this scheme. The verification is an entirely original effort.

I formalised three languages in the proof assistant Agda: a conventional language called LET, an intermediate language ML_Π and most importantly, a reversible language Π. I implemented the translations T_1 from LET to ML_Π and T_2 from ML_Π to Π. ■ denotes proofs: I formally verified the correctness of both T_1 and T_2 and showed that Π is reversible. The dashed box denotes the Turing-complete extension of Π, known as Π°, and □ denotes the proof of its reversibility.

1.1 What is a Reversible Language?

For all programs written in such a language, we can recover the input given the output. We can make a program trivially reversible by just returning the input along with the output but that would not help the main use-case of reversible computing i.e energy-efficiency. For this purpose, ideally **every** step must be performed reversibly. Π, having been built up from category-theoretic isomorphisms, satisfies the latter criteria.

1.2 Motivations and Related Work

Why is reversible computing energy-efficient?

Landauer [13] found that during computation, the only operations that result in irreversible energy loss are bit erasures and branch merges. Thus, the energy consumption of a computation is not determined by how much information it involves but rather by how much information

it loses. So a computation that does not lose any information at **any** step, i.e. a reversible computation, can theoretically result in no loss of energy at all.

Why should we compile from LET?

One of the primary technical challenges in this project was to learn how to program in Π using categorical constructs. Trace operators in Π° posed an even greater challenge. Programming reversibly is not natural for humans. We tend to think of only the forward direction of data flow from the input to the output.

It turns out that it is possible to automate the complex and cumbersome bookkeeping required to reversibly execute most constructs in conventional languages. Thus, a compiler such as the one implemented in this project, will allow us to continue writing programs in high-level languages we are used to while implementing them in low-level reversible languages designed for reversible hardware such as the Pendulum microprocessor [19].

Why verify the compiler?

Reversible programs will likely find use in embedded systems, which often have to be energy-efficient. System and program verification is becoming an increasingly important research area as software is getting ever greater responsibility over our everyday lives. A first step towards trustworthy software is ensuring that the compiler does not introduce bugs, hence the need for compiler correctness. Verified compilers are already being built for C (CompCert [14]) and ML (CakeML [12]). Nearer to the reversible computing sphere, there are verifications of quantum computing protocols in Coq [4] [1] and formalisations of quantum algorithms in Isabelle [5] [2]. Reversibility is important in quantum computing since quantum circuits must be reversible.

1.2.1 Overview of Reversible Languages

Π and Π° were introduced in Bowman et al. [21]. They differed from the two most prominent reversible languages at the time, Janus [15] and R [8], in that their design had a solid mathematical basis whereas their precursors were, while elegant, rather *ad-hoc*. Janus, for instance, requires that the programmer provide both an entry and an exit assertion for a loop, so that the exit assertion may be used as the entry condition when executing the aforementioned program in reverse. This shifts some of the burden of reversibility onto the programmer.

Many languages like Π and Π° have since been formalised in Agda [6] to show equivalences between reversible computing and homotopy type theory. I discuss Π further in section 2.1.3.

Chapter 2

Preparation

In this chapter, I first summarise the ideas in James and Sabry [10]. Then I discuss Agda and software development considerations.

2.1 Overview of languages and compilation

2.1.1 High-level overview

The intermediate language ML_Π makes information creation and erasure explicit. Inspired by Toffoli's theorem [18], [10] introduces a heap and garbage during translation from ML_Π to Π. Wherever information may be created, a value is added to the heap. Values that might otherwise be erased are added to the garbage. The wiring and category-theoretic diagrams in the sections that follow are my own additions. I explain Π° in section 3.4.4.

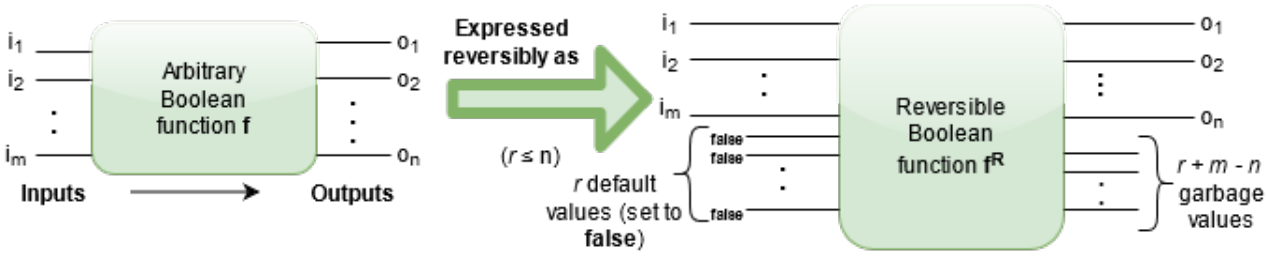


Figure 2.1: Toffoli's theorem proves constructively that we can express any Boolean function reversibly by adding a heap and garbage.

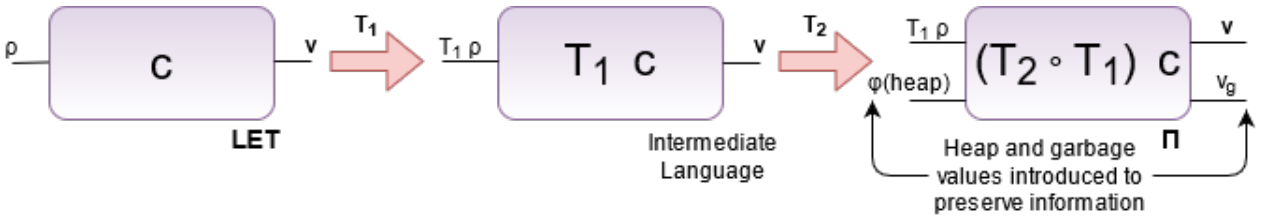


Figure 2.2: Similar to Toffoli, additional inputs and outputs introduced for reversibility

2.1.2 LET - the high-level irreversible language

LET has an entirely conventional structure. Like Π, it is strongly normalising so all constructs terminate. Typing rules are discussed in section 2.1.5 alongside T_1, so here I will present the relevant grammars and representative operational semantics:

Type environments $\Gamma ::= \epsilon \mid \Gamma, x : b$
Value environments $\rho ::= \epsilon \mid \rho, x = v$
Expressions $e ::= () \mid x \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{left } e \mid \text{right } e \mid \text{case } e \text{ in } \mathrm{L}x \to e_1 \text{ , } \mathrm{R}y \to e_2 \mid \text{fst } e \mid \text{snd } e \mid (e, e)$

$$\frac{(e_1,\rho)\Downarrow v_1 \quad (e_2,\rho)\Downarrow v_2}{((e_1{,}e_2),\rho)\Downarrow [v_1{,}v_2]}\ \times_{\mathbf{e}} \qquad \frac{(e_1,\rho)\Downarrow v_1 \quad (e_2,[\rho,\ x=v_1])\Downarrow v_2}{(\text{let } x=e_1 \text{ in } e_2\ ,\rho)\Downarrow v_2}\ \mathbf{let_e} \qquad \frac{(e,\rho)\Downarrow v}{(\text{left } e,\rho)\Downarrow \text{left } v}\ \mathbf{left_e}$$

$$\frac{(e,\rho)\Downarrow \text{right } v \quad (e_2,[\rho,\ y=v])\Downarrow v_2}{(\text{case } e \text{ in } \mathrm{L}x\to e_1\ ,\ \mathrm{R}y\to e_2\ ,\rho)\Downarrow v_2}\ \mathbf{case_{eR}} \qquad \frac{(e,\rho)\Downarrow [v_1{,}v_2]}{(\text{fst } e,\rho)\Downarrow v_1}\ \mathbf{fst_e} \qquad \frac{\rho(x)\Downarrow v}{(x,\rho)\Downarrow v}\ \mathbf{var_e} \qquad \frac{}{()\Downarrow ()}\ \mathbf{1_e}$$

2.1.3 Π - the low-level reversible language

Programs in Π are analogous to invertible category-theoretic transformations known as **isomorphisms**. This ensures that Π is reversible at every step. Because programs must be reversible, i.e. we should be able to recover the input given the output and vice-versa, they should be invertible in both directions. So every program is a bijection between its input type b_1 and output type b_2. Types correspond to category-theoretic objects.

Types

All languages considered here use finite types for values. Finite types are built up from sums and products of the primitive type 1. They are represented by $b ::= 1 \mid b \times b \mid b + b$. For example, $1 + 1$ is a valid type and is used later as the boolean type. The table below draws analogies between types in Π, objects in a cartesian closed category **C** and sets in the category **Set** of sets. The intuition behind the analogy with **Set** is that a type is a set and the values of that type are the elements of that set.

Π	**C**	**Set**
1	Terminal object $\top$	Singleton object $\{\varnothing\}$
$b_1 \times b_2$	Binary product $B_1 \times B_2$	Cartesian product $S_1 \times S_2$
$b_1 + b_2$	Binary coproduct $B_1 + B_2$	Disjoint union $S_1 + S_2$

Table 2.1: Representations of Π types in categories **C** and **Set**

$$\frac{}{\vdash () : 1} \qquad \frac{\vdash v_1 : b_1 \quad \vdash v_2 : b_2}{\vdash [v_1{,}v_2] : b_1 \times b_2} \qquad \frac{\vdash v : b_1}{\vdash \text{left } v : b_1 + b_2} \qquad \frac{\vdash v : b_2}{\vdash \text{right } v : b_1 + b_2}$$

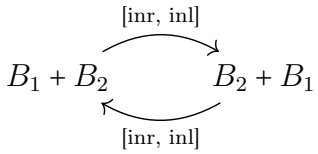
Figure 2.3: Typing rules for Π values

Program constructs or *combinators*

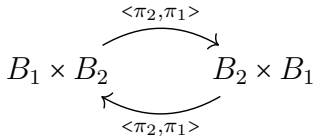
The primitive isomorphisms in Figure 2.4 form the basis of Π's primitive constructs. Their congruence closure, out of which we can derive Π's composite constructs (Figure 2.5), is the 'set of sound and complete isomorphisms for finite types'. [7] [10]

2.1.4 ML_Π - the intermediate language

The implicit heap and garbage are only required when there is an irreversible operation, i.e. the information creation or erasure, which we indentify in LET constructs using ML_Π. ML_Π consists of the same set of primitive isomorphisms as Π, with two additional primitives: $\mathbf{create}_b$ (used to create a default value of type b) and **erase** (used to erase a value of any type). $\mathbf{create}_b$ and **erase** are not isomorphisms. If they were, ML_Π would not be irreversible. I show how the irreversibility in some common constructs can be made explicit using $\mathbf{create}_b$ and **erase**.



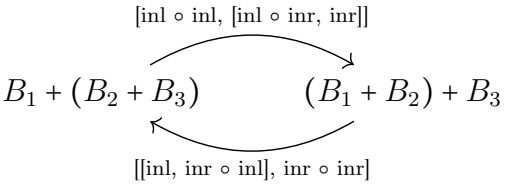
$$\mathbf{swap}^+ : b_1 + b_2 \leftrightarrow b_2 + b_1 : \mathbf{swap}^+$$
$$\mathit{left}\ v \leftrightarrow \mathit{right}\ v$$



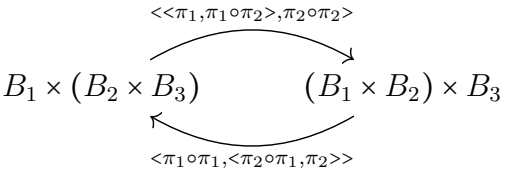
$$\mathbf{swap}^\times : b_1 \times b_2 \leftrightarrow b_2 \times b_1 : \mathbf{swap}^\times$$
$$[v_1,v_2] \leftrightarrow [v_2,v_1]$$

$$1 \times B \underset{\text{<(),id>}}{\overset{\pi_2}{\rightleftarrows}} B$$

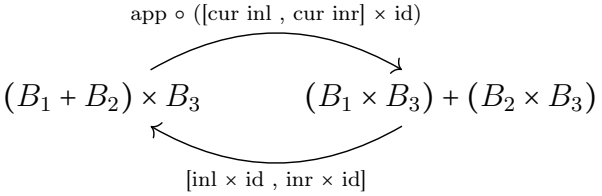
$$\mathbf{unite} : 1 \times b \leftrightarrow b : \mathbf{uniti}$$
$$[(),v] \leftrightarrow v$$



$$\mathbf{assocl}^+ : b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 : \mathbf{assocr}^+$$
$$\mathit{left}\ v_1 \leftrightarrow \mathit{left}\ (\mathit{left}\ v_1)$$
$$\mathit{right}\ (\mathit{left}\ v_2) \leftrightarrow \mathit{left}\ (\mathit{right}\ v_2)$$
$$\mathit{right}\ (\mathit{right}\ v_3) \leftrightarrow \mathit{right}\ v_3$$



$$\mathbf{assocl}^\times : b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 : \mathbf{assocr}^\times$$
$$[v_1,[v_2,v_3]] \leftrightarrow [[v_1,v_2],v_3]$$



$$\mathbf{distrib} : (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) : \mathbf{factor}$$
$$[\mathit{left}\ v_1,v_3] \leftrightarrow \mathit{left}\ [v_1,v_3]$$
$$[\mathit{right}\ v_2,v_3] \leftrightarrow \mathit{right}\ [v_2,v_3]$$

Figure 2.4: Primitive operators in Π

$$B_1 \underset{\text{f}^{-1}}{\overset{\text{f}}{\rightleftarrows}} B_3 \qquad B_2 \underset{\text{g}^{-1}}{\overset{\text{g}}{\rightleftarrows}} B_4$$

$$B_1 \times B_2 \underset{\text{f}^{-1}\times\text{g}^{-1}}{\overset{\text{f}\times\text{g}}{\rightleftarrows}} B_3 \times B_4$$

$$\frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_3 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4}\ \times$$

$$\frac{c_1\ v_1 \mapsto v_3 \quad c_2\ v_2 \mapsto v_4}{c_1 \times c_2\ [v_1,v_2] \mapsto [v_3,v_4]}\ \text{eval}_\times$$

$$B_1 \underset{\text{f}^{-1}}{\overset{\text{f}}{\rightleftarrows}} B_3 \qquad B_2 \underset{\text{g}^{-1}}{\overset{\text{g}}{\rightleftarrows}} B_4$$

$$B_1 + B_2 \underset{\text{f}^{-1}+\text{g}^{-1}}{\overset{\text{f+g}}{\rightleftarrows}} B_3 + B_4$$

$$\frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_3 \leftrightarrow b_4}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4}\ +$$

$$\frac{c_1\ v_1 \mapsto v_3}{c_1 + c_2\ (\mathit{left}\ v_1) \mapsto (\mathit{left}\ v_3)}\ \text{eval}_{L+}$$

$$\frac{c_2\ v_2 \mapsto v_4}{c_1 + c_2\ (\mathit{right}\ v_2) \mapsto (\mathit{right}\ v_4)}\ \text{eval}_{R+}$$

$$B \underset{\text{id}}{\overset{\text{id}}{\rightleftarrows}} B$$

$$\frac{}{\text{id} : b \leftrightarrow b}\ \text{id}$$

$$\frac{}{\text{id}\ v \mapsto v}\ \text{eval}_{\text{id}}$$

$$B_1 \underset{\text{f}^{-1}}{\overset{\text{f}}{\rightleftarrows}} B_2 \underset{\text{g}^{-1}}{\overset{\text{g}}{\rightleftarrows}} B_3 \qquad \text{g} \circ \text{f} \qquad \text{f}^{-1} \circ \text{g}^{-1}$$

$$\frac{c_1 : b_1 \leftrightarrow b_2 \qquad c_2 : b_2 \leftrightarrow b_3}{c_1;c_2 : b_1 \leftrightarrow b_3}\ ;$$

$$\frac{c_1\ v_1 \mapsto v_2 \qquad c_2\ v_2 \mapsto v_3}{c_1;c_2\ v_1 \mapsto v_3}\ \text{eval}_{;}$$

$$B_1 \underset{\text{f}^{-1}}{\overset{\text{f}}{\rightleftarrows}} B_2$$

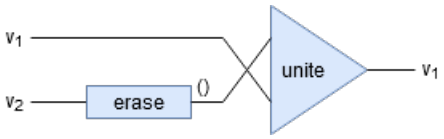
$$\frac{c : b_1 \leftrightarrow b_2}{sym\ c : b_2 \leftrightarrow b_1}\ \text{id}$$

$$\frac{c^{\dagger}\ v_1 \mapsto v_2}{(sym\ c)\ v_1 \mapsto v_2}\ \text{eval}_{\text{sym}}$$

Figure 2.5: Composition combinators in Π

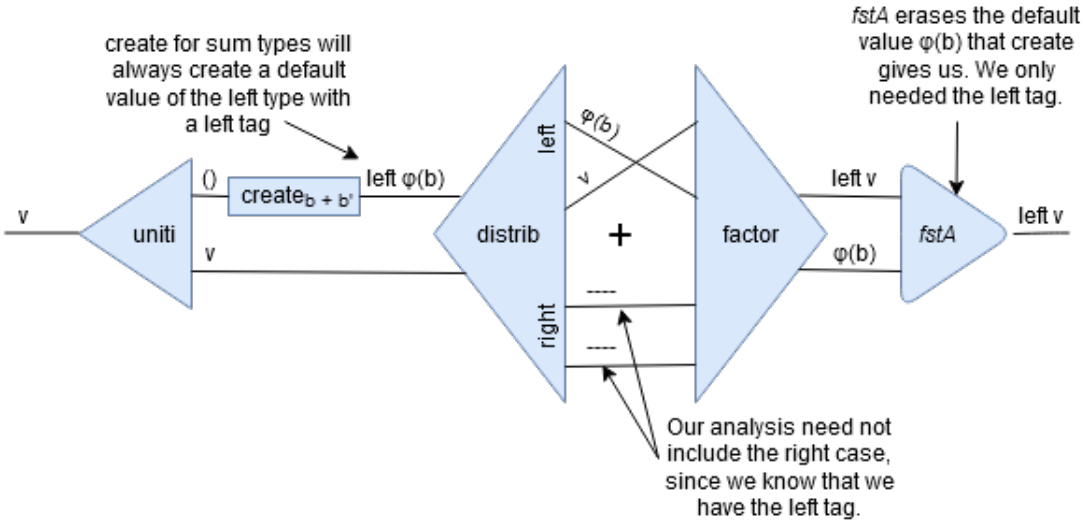
ML_{Π} Example combinators

fstA: Consider an operator that takes the first value from a pair. The second value is erased, which we can make explicit in the representation below.



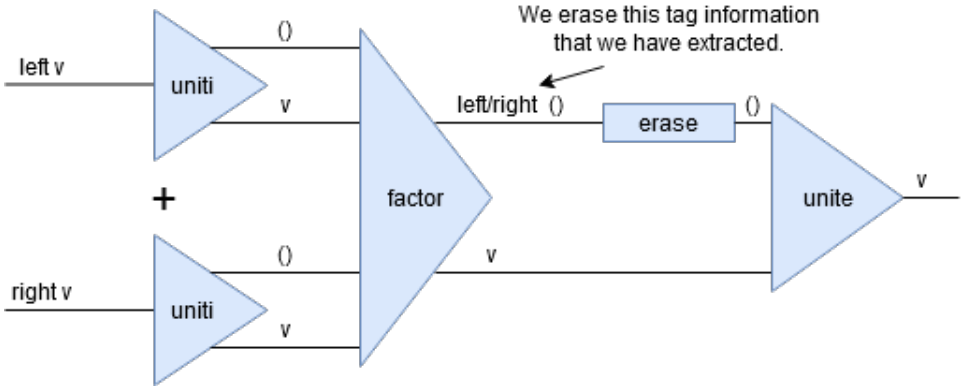
Note for understanding the wiring diagrams: These are in the style of [10]. Wires represent values, and multiple wires run in parallel if their corresponding values exist in the system simultaneously. We may abstract away the pairing of values since pairs are often just used to represent tuples. A + sign between wires represents a sum type. Occasionally triangles will be used to represent combinators such as **unite** or **distrib**. However, for the simpler combinators, we do not use triangles. This is because many of these combinators are used to move values around, which we can show via crossing of wires. For example, **swap**$^{\times}$, which exchanges two values, can be represented by crossing the two representative wires.

leftA: This operator takes a value v of type b, and attaches a left tag to it. We are adding arbitrary information, so we need to call **create**.



join: Another important operator takes a tagged value of type $b+b$, and removes the tag. This change is also irreversible. If we are presented with an output value v of type b, it is impossible

to tell whether the input value was *left v* or *right v*. **join** exposes the inherent irreversibility in branching or case statements, where we get a value of the same type from all possible branches and it may be impossible to tell which branch was taken.



Syntax and semantics

ML_Π uses the same type theory for values as LET and Π. The set a of ML_Π combinators is a superset of *iso*, the set of combinators in Π. Combinators can have two types: $b \leftrightarrow b$ or $b \rightsquigarrow b$. The former is, as discussed in section 2.1.3, the typing syntax for *iso*. The latter is the typing syntax for the (potentially) irreversible constructs in ML_Π. I present the associated typing rules and operational semantics.

$$\frac{c : b_1 \leftrightarrow b_2}{arr\ \ c : b_1 \rightsquigarrow b_2}\ \textbf{arr} \qquad \frac{c\ v_1 \leftrightarrow v_2}{(arr\ \ c)\ v_1 \rightsquigarrow v_2}\ \textbf{arr}_\textbf{e}$$

$$\frac{a_1 : b_1 \rightsquigarrow b_2 \quad a_2 : b_2 \rightsquigarrow b_3}{a_1 \ggg a_2 : b_1 \rightsquigarrow b_3}\ \ggg \qquad \frac{a_1\ v_1 \rightsquigarrow v_2 \quad a_2\ v_2 \rightsquigarrow v_3}{(a_1 \ggg a_2)\ v_1 \rightsquigarrow v_3}\ \ggg_\textbf{e}$$

$$\frac{a : b_1 \rightsquigarrow b_2}{first\ \ a : b_1 \times b_3 \rightsquigarrow b_2 \times b_3}\ \textbf{first} \qquad \frac{a\ v_1 \rightsquigarrow v_2}{(first\ \ a)\ [v_1,v_3] \rightsquigarrow [v_2,v_3]}\ \textbf{first}_\textbf{e}$$

$$\frac{}{create_b : 1 \rightsquigarrow b}\ \textbf{create} \qquad \frac{}{create_b\ ()\rightsquigarrow \phi(b)}\ \textbf{create}_\textbf{e}$$

$$\frac{a : b_1 \rightsquigarrow b_2}{left\ \ a : b_1 + b_3 \rightsquigarrow b_2 + b_3}\ \textbf{left} \qquad \frac{a\ v_1 \rightsquigarrow v_2}{(left\ \ a)\ left\ v_1 \rightsquigarrow left\ v_2}\ \textbf{left}_\textbf{eL} \qquad \frac{}{(left\ \ a)\ right\ v \rightsquigarrow right\ v}\ \textbf{left}_\textbf{eR}$$

$$\frac{}{erase : b \rightsquigarrow 1}\ \textbf{erase} \qquad \frac{}{erase\ v \rightsquigarrow ()}\ \textbf{erase}_\textbf{e}$$

Figure 2.6: Typing rules and semantics for ML_Π irreversible constructs

The **default value** $\phi(b)$ is defined inductively:

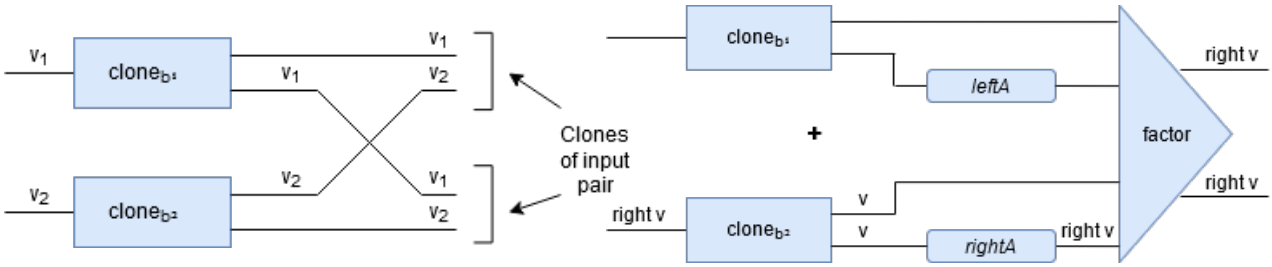
$$\phi(1) = () \qquad \phi(b_1 \times b_2) = [\phi(b_1),\phi(b_2)] \qquad \phi(b_1 + b_2) = left\ \phi(b_1)$$

Cloning

Cloning is analogous to *fan-out* in physical circuits and we do not erase any information. However, while logically reversible, it is not possible in some reversible models of computation (cf. no-cloning theorem in quantum computing). We will need cloning to translate constructs in LET where we must provide copies of the environment to various sub-expressions.

We cannot express cloning in Π. This would have been possible were our set of types limited to the unit primitive and products, because in that case, there would be just one value of a particular type. However, this is no longer true when we incorporate sum types. Consider what happens when we try to run a cloning circuit in reverse on the input [*left* v,*right* v]. No output exists. This shows that cloning combinators for sum types are not isomorphisms, and since we define $\mathbf{clone}_b$ by induction, we cannot say that $\mathbf{clone}_{b_1 \times b_2}$ (where $b_1 \times b_2$ is a product type) is an isomorphism either.

$\mathbf{clone}_1$ is just **uniti**, which takes () to [(),()]. The following figure shows how cloning combinators are defined for sum and product types:



(a) Cloning for pair [v_1,v_2]

(b) Cloning for *right* v

Figure 2.7: Wiring diagrams for $\mathbf{clone}_b$

2.1.5 T_1 - the translation from LET to ML_Π

When we evaluate an expression in LET, we are left with a value. The hidden input here is the environment. T_1 makes this fact explicit by translating LET expressions to ML_Π combinators and environments to tuples, which are encoded using products. We can summarize this translation as follows:

$$(e, \rho)_{\mathbf{LET}} \longrightarrow (\mathbf{T}_1\ e)_{\mathbf{ML}_\Pi}\ (\mathbf{T}_1\ \rho)_{\mathbf{ML}_\Pi}$$

A LET typing judgement $\Gamma \vdash e : b$ is thus translated to an ML_Π combinator $a : \Gamma^\times \rightsquigarrow b$, where $\Gamma^\times$ is the translation of the type environment Γ. Evaluating e in an environment ρ results in the same value as evaluating a on a translation of ρ. I discuss a formal proof of this in section 3.3.3. Below I describe the translation for each LET construct:

$$\frac{}{\Gamma \vdash () : 1}$$

Regardless of the environment, the expression () will always evaluate to the same value. So we must be **erasing** all the information in the environment. T_1 translates this typing judgement to **erase** : $\Gamma^\times \rightsquigarrow \mathbf{1}$.

Here the environment is used twice, once for evaluating e_1 to assign to x and then for evaluating e_2 in an extension of it. e_1 may arbitrarily add or remove information from the environment, rendering it unusable for e_2. So we need two copies of the environment and thus,

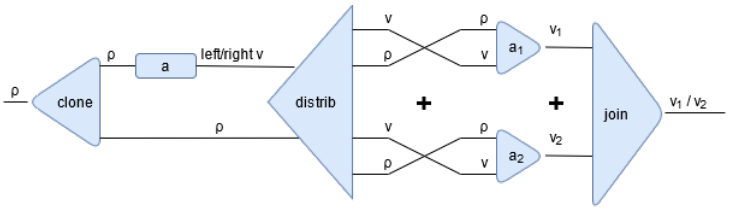
$$\frac{\Gamma \vdash e_1 : b_1 \quad \Gamma, x : b_1 \vdash e_2 : b_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : b_3}$$

let in ML_Π

Figure 2.8: The **let** construct: this allows us to express functions.

this translation requires **cloning**. We will pass one copy to $a_1 : \Gamma^\times \rightsquigarrow b_1$, the translation of $\Gamma \vdash e_1 : b_1$. This will give us the value of x that we need. We extend our other copy of the environment with this value, and pass this extended environment to $a_2 : \Gamma^\times \times b_1 \rightsquigarrow b_2$, the translation of $\Gamma, x : b_1 \vdash e_2 : b_2$. Our translation is now complete.

$$\frac{\Gamma \vdash e : b_1 + b_2 \quad \Gamma, x : b_1 \vdash e_1 : b_3 \quad \Gamma, y : b_2 \vdash e_2 : b_3}{\Gamma \vdash \text{case } e \text{ in } \text{L}x \to e_1 \text{ , } \text{R}y \to e_2 \text{: } b_3}$$



case in ML_Π

Figure 2.9: The **case** construct for branching

The translation of **case** expressions is similar to **let** above in that we construct two copies of the environment. From passing a copy of the environment to $a : \Gamma^\times \rightsquigarrow b_1 + b_2$, the translation of $\Gamma \vdash e : b_1 + b_2$, we get a value v of the sum type $b_1 + b_2$. Depending on whether v has a *left* or a *right* tag, we will be able to extend our copy of the environment with a value of type b_1 or b_2. If v has a *right* tag, then we can pass our extended environment to $a_2 : \Gamma^\times \times b_2 \rightsquigarrow b_3$, the translation of $\Gamma, y : b_2 \vdash e_2 : b_3$. Similarly, if v has a *left* tag, we pass to $a_1 : \Gamma^\times \times b_1 \rightsquigarrow b_3$,, the corresponding translation for $\Gamma, x : b_1 \vdash e_1 : b_3$. The output from the left and right branches is tagged with the direction of the branch, and we use **join** to irreversibly remove this tag.

For pairs **(**e_1**,**e_2**)**, we must again evaluate e_1 and e_2 in identical environments. Therefore, we clone the environment and pass one copy to $a_1 : \Gamma^\times \rightsquigarrow b_1$, the translation of $\Gamma \vdash e_1 : b_1$ and another copy to $a_2 : \Gamma^\times \rightsquigarrow b_2$, the translation of $\Gamma \vdash e_2 : b_2$ to get our required pair of values.

$$\frac{\Gamma \vdash e_1 : b_1 \quad \Gamma \vdash e_2 : b_2}{\Gamma \vdash (e_1,e_2) \text{: } b_1 \times b_2} \qquad \frac{\Gamma \vdash e : b_1 \times b_2}{\Gamma \vdash \mathit{fst}\ e \text{: } b_1} \qquad \frac{\Gamma \vdash e : b_1}{\Gamma \vdash \mathit{left}\ e \text{: } b_1 + b_2}$$

Figure 2.10: Typing rules for pairs, **fst** and **left** - **snd** and **right** are typed similarly

For **fst** e and **snd** e, we simply evaluate e using $a : \Gamma^\times \rightsquigarrow b_1 \times b_2$, the translation of $\Gamma \vdash e : b_1 \times b_2$, and then use **fstA** and **sndA** respectively to irreversibly remove one value from the pair.

As for **left** e and **right** e, similar to **fst** and **snd** above, we evaluate e using $a : \Gamma^\times \rightsquigarrow b_1 + b_2$, the translation of $\Gamma \vdash e : b_1 + b_2$, and then use **leftA** and **rightA** to add tag information to the value that we get.

I conclude my description of T_1 by describing how it translates environments. Both type environments Γ and value environments ρ are represented as tuples, which we encode using pairs. The translation of Γ is denoted by the type $\Gamma^\times$ and the translation of ρ is denoted by the value $\rho^\times$. Both an empty type environment and an empty value environment are denoted by ϵ.

For the translation of type environments:

$$\epsilon^\times \longrightarrow 1$$
$$[\Gamma, x : b]^\times \longrightarrow \Gamma^\times \times b$$

And for value environments:

$$\epsilon^\times \longrightarrow ()$$
$$[\rho, x = v]^\times \longrightarrow (\rho^\times, v)$$

Finally, a typing rule for variable references:

$$\frac{\Gamma(x) = b}{\Gamma \vdash x : b}$$

In section 3.3.1, I will present a combinator $\mathbf{lookup}_x$ that looks x up in the translated environment $\Gamma^\times$. We translate any reference to x in an expression to $\mathbf{lookup}_x$.

2.1.6 T_2 - the translation from ML_Π to Π

Each ML_Π combinator explicitly interacts with the external environment via **create** (where it takes a value from an implicit heap) and **erase** (where it puts a value into an implicit garbage). T_2 makes the heap and garbage explicit. Consider an ML_Π combinator $c : b_1 \rightsquigarrow b_2$. We determine an appropriate heap type h and garbage type g for c and output a Π combinator $(T_2\ c)$:

$$(c)_{\mathbf{ML}_\Pi} : b_1 \rightsquigarrow b_2 \longrightarrow (\mathbf{T}_2\ c)_\Pi : h \times b_1 \leftrightarrow g \times b_2$$

We initialize the heap with default value $\phi(h)$, where $\phi(b)$ is the value returned by $\mathbf{create}_b$. We also pass in the original argument v. Evaluating $(\mathbf{T}_2\ c)$ on these two arguments will return $(c\ v)_{\mathbf{ML}_\Pi}$ and a garbage value v_g that contains all the information thrown away during the computation of $(c\ v)_{\mathbf{ML}_\Pi}$.

The most obvious cases are $\mathbf{create}_b$ and **erase**. $\mathbf{create}_b$, given the unit as input, outputs the default value $\phi(b)$ of type b which it obtains from a heap of type b. There is no garbage produced and we denote an empty garbage using the type 1. Thus, we translate $\mathbf{create}_b$ to a combinator $(\mathbf{T}_2\ \mathbf{create}_b)$: $b \times 1 \leftrightarrow 1 \times b$. The input to $(\mathbf{T}_2\ \mathbf{create}_b)$ is $[\phi(b), ()]$ and the output is $[(), \phi(b)]$, so $(\mathbf{T}_2\ \mathbf{create}_b)$ is simply $\mathbf{swap}^\times$.

Likewise, **erase** takes a value v of any type b, moves it into the garbage (which must therefore be of type b) and returns the value $()$ of type 1. The heap can thus be empty i.e. of type 1. So we translate **erase** to $(\mathbf{T}_2\ \mathbf{erase})$: $1 \times b \leftrightarrow b \times 1$, which takes $[(), v]$ and returns $[v, ()]$. This shows that $(\mathbf{T}_2\ \mathbf{erase})$ is also $\mathbf{swap}^\times$.

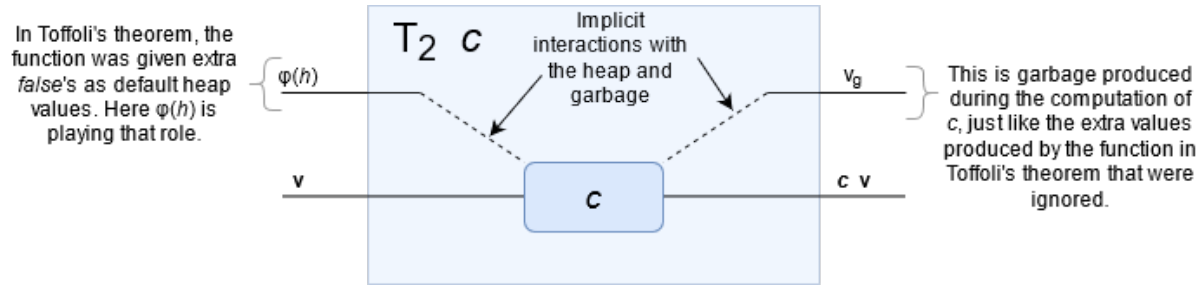


Figure 2.11: Wiring diagram representing T_2

Now we move to the ML_Π constructs that are composed of smaller constructs. First we start with **arr** c, which is equivalent to the Π construct c and is thus reversible. So both the heap and garbage are empty and of type 1. If **arr** c: $b_1 \rightsquigarrow b_2$, then $\mathbf{T}_2$ **(arr** c**)**: $1 \times b_1 \leftrightarrow 1 \times b_2$ and is equal to **id** $\times$ c.

The sequential combinator $(a_1 \ggg a_2)$ that composes a_1 and a_2 is more difficult to translate. Suppose we have already translated a_1 and a_2, and say we need default heap values $\phi(h_1)$ and $\phi(h_2)$ for their computations respectively, i.e. h_1 is the heap type for a_1 and similarly h_2 is the heap type for a_2. Within the computation of $(a_1 \ggg a_2)$, we will evaluate both a_1 and a_2 at some point, so the heap value we pass in must contain both $\phi(h_1)$ and $\phi(h_2)$. The computations will produce garbage values v_{g^1} of type g^1 and v_{g^2} of type g^2 respectively. So the garbage we output as a result of $(a_1 \ggg a_2)$'s computation must contain at least these values.

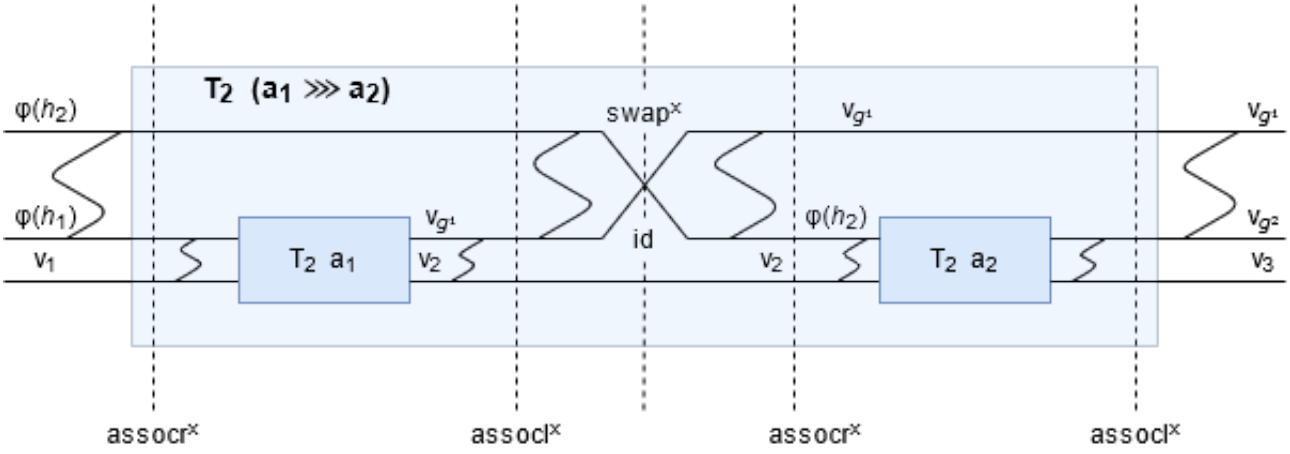


Figure 2.12: Wiring diagram representing $\mathbf{T}_2$ $\mathbf{(a_1 \ggg a_2)}$

It so happens that we do not need extra values in the heap or garbage aside from those discussed. The heap type we use is $h_2 \times h_1$ and the garbage type is $g^1 \times g^2$. The heap can also be of type $h_1 \times h_2$ and the garbage of type $g^2 \times g^1$, depending on how we move the values around. The given heap and garbage types correspond to James and Sabry's proposed combinator for $\mathbf{T}_2$ $(a_1 \ggg a_2)$. First we evaluate $\mathbf{T}_2$ a_1 on our input value v_1 and the heap value $\phi(h_1)$, which produces an intermediate value v_2 (corresponding to $(a_1\ v_1)_{\mathbf{ML}_\Pi}$) and a garbage value v_{g^1}. We then evaluate $\mathbf{T}_2$ a_2 on v_2 and the heap value $\phi(h_2)$, which produces our final value v_3 and another garbage value v_{g^2}. The wiring diagram above shows how the values are moved around in the computation. Translating the movements in the wiring diagram to combinators, we get:

$$\mathbf{T}_2\ (a_1 \ggg a_2) = \mathbf{assocr}^\times; \mathbf{id} \times (\mathbf{T}_2\ a_1); \mathbf{assocl}^\times; \mathbf{swap}^\times \times \mathbf{id}; \mathbf{assocr}^\times; \mathbf{id} \times (\mathbf{T}_2\ a_2); \mathbf{assocl}^\times$$

Now we consider **first** a. It only involves the computation of a on the first value in a pair while ignoring the second. For this computation, we must provide the heap value for a's computation, i.e. $\phi(h)$ where h is a's heap type. We will get a garbage value v_g where g is a's garbage type. Thus, $\mathbf{T}_2$ **(first** a**)**: $h \times (b_1 \times b_2) \leftrightarrow g \times (b_3 \times b_2)$, where a: $b_1 \rightsquigarrow b_2$. Writing it in terms of combinators is fairly straightforward: $\mathbf{T}_2$ **(first** a**)** = **assocl**$^{\times}$; **(**$\mathbf{T}_2$ a**)** $\times$ **id**; **assocr**$^{\times}$.

$$\frac{a : b_1 \rightsquigarrow b_2 \longrightarrow^{T_2} h \times b_1 \leftrightarrow g \times b_2}{(\mathbf{left}\ a) : b_1 + b_3 \rightsquigarrow b_2 + b_3 \longrightarrow^{T_2} (h \times ((b_2 + b_3) + (b_3 + b_2))) \times (b_1 + b_3) \leftrightarrow (g' + g'') \times b_2 + b_3}$$

$$g' = g \times (b_2 \times (b_3 + b_2)) \ \mathbf{and}\ g'' = (h \times (b_2 + b_3)) \times b_3$$

Dealing with computations on sum types turns out to be extremely non-trivial. Above I simply present the heap and garbage types for **left** a as laid out in [10]. I present a justification for this and then derive the recursive equation for $\mathbf{T}_2$ **(left** a**)** in section 3.5, where I use it as a starting point to translate $(a_1 \oplus a_2)$.

2.2 Software development tools and strategies

My supervisor chose the dependently-typed language Agda [16] for this project. I used agda-mode on GNU Emacs to code, mostly because there are convenient commands to, for instance, convert to normal form or display assumptions. For version control, I used OneDrive for this dissertation and GitHub for the code. A brief introduction to Agda follows.

Dependent types can refer to or 'depend on' values. $\mathbb{b}$ represents [Π] types and could be similarly represented in ML as the recursive data type $\mathbb{b} = \mathbb{1} \mid + \ \mathbf{of}\ \mathbb{b}\ \mathbf{*}\ \mathbb{b} \mid \times\ \mathbf{of}\ \mathbb{b}\ \mathbf{*}\ \mathbb{b}$. Values are indexed by their types which are represented as [Agda] values of $\mathbb{b}$. Hence, the types of [Π] values 'depend' on [Agda] values of $\mathbb{b}$, resulting in an implicit typing judgement whenever a value is constructed. For example, given a value v of type b_1, **left** v must have type $b_1 + b_2$ for arbitrary b_2:

```
data 𝕓 : Set where
  𝟙 : 𝕓
  _×_ _+_ : 𝕓 → 𝕓 → 𝕓
```

```
data val : 𝕓 → Set where
  [] : val 𝟙
  left : ∀{b₁ b₂}
       → val b₁
       ------
       → val (b₁ + b₂)
```

The types Fin n and Vec A n are indexed by the natural numbers. Fin n is the type of all natural numbers less than n while Vec A n is a parameterised type representing a vector of length n containing objects of type A. The elements in a vector of length n are numbered from 0 to $n-1$, i.e. exactly the elements of Fin n. Thus, we can ensure that we never get an index-out-of-bounds exception if we use values of type Fin n as indices to access a vector of type Vec A n.

Proofs and termination

By the Curry-Howard correspondence, we can represent propositions as Agda types and their proofs as the terms of those types. The proof of $\forall x : X.P(x)$ is a dependent function f which returns a proof of $P(x)$ given a proof of x. For example, the ML_{Π} interpreter, for any

combinator c and value of c's input type b, returns a value of c's output type b', giving us an **implicit type-safety proof** for ML_Π:

```
_[_]ᵃ : ∀ {b b' : 𝔹} → b ⇝ b' → val b → val b'
```

Agda is constructive, so to prove $\exists x \in X.P(x)$, we must provide a witness x and a proof of $P(x)$, whose type depends on the witness. T_2's verification is an existence proof. Save for dependent types, Agda is similar to ML and Haskell, except that all functions in Agda must be total and terminate on all inputs since non-termination introduces logical inconsistency.

The type $a \equiv b$ represents the propositional equality of terms a and b. If Agda can reduce one term to the other, then reflexivity, written refl, will suffice as proof of the equality. Properties such as symmetry, congruence and transitivity can also be proved and there is syntax for equational reasoning. Since terms in my proofs are generally very involved, I use the **rewrite** syntax, as explained in section 3.3.3. Merits and demerits of my approach are discussed in section 4.3.

Starting Point

I experimented with Agda basics over the summer but have no other relevant knowledge save for the Category Theory unit and *Compiler Construction* course.

Software Development Life Cycle

The notion of test-driven development with a theorem-prover is slightly odd, since programs are type-checked and verified to some extent at compilation. Regardless, I met the following requirements sequentially, with alternating formalisations and proofs. The strategy of implementing the core and extension separately was not a good idea and is a rare failure of the iterative development process. Π° required a different representation of types.

Requirement	Priority	Difficulty
Values and Types	high	low
Π syntax and semantics	high	medium
Proof of Π's reversibility	medium	high
ML_Π syntax and semantics	high	low
ML_Π example combinators	high	medium
Verification of cloning in ML_Π	high	low
LET syntax and semantics	high	high
T_1 implementation	high	medium
T_1 verification	medium	high
T_2 implementation	high	high
T_2 verification	medium	medium
Extension: Π°	low	high
Extension: Turing-completeness of Π°	low	high

Table 2.2: Requirements analysis with priority

Chapter 3

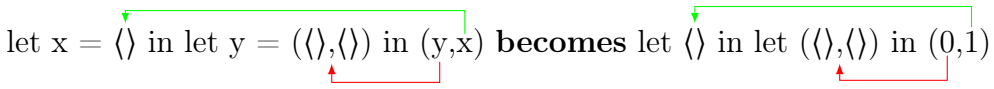
Implementation

Most of this chapter is an original effort. There were six major challenges: the data structures to represent LET environments and expressions in Agda, formalising the proof of T_1, extending the ideas of James and Sabry to carry out T_2 on the ML_Π construct $\oplus$, the declaration of the proof of T_2 and lastly, proving the reversibility of Π (which my supervisor has helped me with), and lastly, learning to program in Π°.

3.1 LET - the conventional high-level irreversible language

3.1.1 Variables, environments and de Bruijn indices

Variable names are pointers to variable bindings in the environment. A de Bruijn index [3] is an alternate way of referring to a binding using its level, with the innermost level being 0. So I replace variable names with natural numbers and avoid additional code complexity:



However, we cannot allow just any natural number to be used in place of a variable name. This can lead to errors during evaluation. In the example above, if 0 is replaced with 5, we will get an error because there is no binding at level 5! In other words, this variable is not in scope. The data structures used to represent indices and environments help us prevent this problem.

```
data _env : ∀{n : ℕ} → (Vec 𝕓 n) → Set where
  ϵ : [] env
  _+ₑ_ : ∀{n : ℕ}{Γ : Vec 𝕓 n}{b : 𝕓}
           → Γ env
           → (x : val b)
         -----
           → (b :: Γ) env
```

(a) Data type for environments

```
varₑ : ∀{n : ℕ}{b : 𝕓}
         {Γ : Vec 𝕓 n}{x : Fin n}
       → Γ [ x ]= b
       -----
       → Γ ⊢exp: b
```

(b) Constructor for variables

Figure 3.1: Variables and environments

Consider a LET value environment ρ and its corresponding type environment Γ. ρ is a vector of values, Γ is a vector of value types, and $\forall x \in [0, ..., \text{len}(\rho) - 1].\rho[x]$ is of type $\Gamma[x]$. In my implementation of LET, I represent ρ as a value of type Γ env, where Γ is a length-indexed vector that holds values of type $\mathbb{b}$, i.e., Γ is of type Vec $\mathbb{b}$ n and n = len(Γ). In this way, I 'connect' ρ to its type environment Γ.

The type environment Γ for any LET expression is known at compile time. Each binding in the current environment contributes one element to Γ. The Agda term $\text{var}_e\ x$ represents a LET variable where x corresponds to a de Bruijn index, and the typechecker ensures that x is of type Fin n where $n = \text{len}(\Gamma)$. Therefore, x must be in the range [0 ,.., len(Γ) - 1] and var_e x refers to a valid binding. By using Fin n instead of $\mathbb{N}$ for x, I prevent references to variables that are out of scope.

3.1.2 The type of an expression is its typing judgement.

Instead of implementing LET typing rules separately, which might lead to expressions that are untypeable, I incorporate the typing derivation into the construction of an expression. This is especially useful when the preconditions are linked in some way, for instance in **case** statements, where the left and right branches must have the same type under the extended environment. So for expressions, I use a data type indexed by value types $\mathbb{b}$ and Γ i.e., Vec $\mathbb{b}$ n. A LET expression **e** is represented in Agda as a value of type $\Gamma \vdash\text{exp:}\ b$ where $\Gamma \vdash \mathbf{e} : b$ according to LET typing rules. Consider the typing rules for **fst** and **case**.

$$\frac{\Gamma \vdash e : b_1 \times b_2}{\Gamma \vdash \text{fst}\ e : b_1}\ \text{fst}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : b_1 + b_2 \\ \Gamma, x : b_1 \vdash e_1 : b_3 \\ \Gamma, y : b_2 \vdash e_2 : b_3\end{array}}{\Gamma \vdash \text{case}\ e\ \text{in L}x \to e_1\ ,\ \text{R}y \to e_2 \text{:}\ b_3}\ \text{case}$$

And the corresponding Agda constructors:

```
fst_e : ∀{n : ℕ}{b₁ b₂ : 𝕓}{Γ : Vec 𝕓 n}
      → Γ ⊢exp: (b₁ × b₂)
      -----
      → Γ ⊢exp: b₁
```

```
_ecase_eL_eR_ : ∀{n : ℕ}{b₁ b₂ b₃ : 𝕓}
                {Γ : Vec 𝕓 n}
  → Γ ⊢exp: (b₁ + b₂)
  → (b₁ :: Γ) ⊢exp: b₃
  → (b₂ :: Γ) ⊢exp: b₃
  -----
  → Γ ⊢exp: b₃
```

Figure 3.2: Construction of **fst** and **case** expressions using typing rules

When we use an expression somewhere, say in an interpreter, Γ is known. To form a typeable **case** expression which is used for branching, we need a LET expression e and a proof that $\Gamma \vdash \mathbf{e} : b_1 + b_2$ for some b_1, b_2. In Agda, this proposition is implicit in the type of e, which is its proof. Similarly, we receive e_1 and e_2 which are proofs that they have the same type b_3 in environments extended with b_1 and b_2 respectively. The typechecker concludes that the **case** expression we have built will have the type b_3 in Γ.

This representation has another big advantage, described in the next section.

3.1.3 A type-safety proof in the declaration of the interpreter

```
eval_e : ∀{n : ℕ}{Γ : Vec 𝕓 n}{b : 𝕓}
       → Γ env → Γ ⊢exp: b → val b
```

LET is a total language so all its constructs will terminate. This fact is verified by Agda's termination checker which ensures that the interpreter eval_e terminates. In addition, we know

from the declaration that it terminates with a value of the correct type **b** given a type environment Γ and an expression e which has type **b** in Γ. So we have shown that LET is type-safe simply by defining eval_e! The definition of eval_e is conventional. I conclude my discussion of LET with two examples:

$$\text{eval}_e\ \rho\ (\text{left}_e\ e) = \text{left}\ (\text{eval}_e\ \rho\ e)$$
$$\text{eval}_e\ \rho\ ({}_e\text{let}\ e_1\ {}_e\text{in}\ e_2) = \text{eval}_e\ (\rho\ +_e\ (\text{eval}_e\ \rho\ e_1))\ e_2$$

3.2 ML_Π - the intermediate language

ML_Π implementation was straightforward. I extended it with the constructs $\oplus$ and $\otimes$, whose typing rules can be described via the constructors for the ML_Π combinator type $\rightsquigarrow$:

$$\begin{array}{l} \_\oplus\_ : \forall\{b_1\ b_2\ b_3\ b_4\} \\ \quad \to b_1 \rightsquigarrow b_2 \\ \quad \to b_3 \rightsquigarrow b_4 \\ \quad \text{--------} \\ \quad \to (b_1 + b_3) \rightsquigarrow (b_2 + b_4) \end{array} \qquad \begin{array}{l} \_\otimes\_ : \forall\{b_1\ b_2\ b_3\ b_4\} \\ \quad \to b_1 \rightsquigarrow b_2 \\ \quad \to b_3 \rightsquigarrow b_4 \\ \quad \text{--------} \\ \quad \to (b_1 \times b_3) \rightsquigarrow (b_2 \times b_4) \end{array}$$

As discussed in section 2.2, $\_[\_]^a$ gives us an implicit type-safety proof for ML_Π and was implemented without deviations from the semantics as laid out in section 2.1.4. In addition to the **leftA**, **join** and **clone**, I also defined **rightA** and **sndA**, whose semantics are shown below and which will be used in T_1:

$$\text{sndA-proof} : \forall\{b_1\ b_2\} \to \forall\{v_1 : \text{val}\ b_1\} \to \forall\{v_2 : \text{val}\ b_2\} \to \text{sndA}\ [\ [\ v_1\ ,\ v_2\ ]\ ]^a \equiv v_2$$
$$\text{rightA-proof} : \forall\{b_1\ b_2\} \to \forall\{v : \text{val}\ b_2\} \to (\text{rightA}\ \{b_1\}\ \{b_2\})\ [\ v\ ]^a \equiv \text{right}\ v$$

The proof of correctness of **clone** was simple and required induction on v:

$$\text{clone-proof} : \forall\{b\} \to \forall(v : \text{val}\ b) \to (\text{clone}\ b)\ [\ v\ ]^a \equiv [\ v\ ,\ v\ ]$$

3.3 T_1 - the translation from LET to ML_Π

The ML_Π combinators required for the translation of each LET construct, are precisely laid out in [10], but the proof of correctness for T_1, on the other hand, was quite difficult and required use of rather uncommon Agda constructs. But first I explain the translation of environments, which differs very slightly from the paper due to my use of de Bruijn indices to implement variables.

3.3.1 Translation of environments: $(\rho : \Gamma)_{\mathbf{LET}} \longrightarrow (\rho_{\mathbf{e}}{}^{\times} : \Gamma^{\times})$

The LET type environment Γ is represented as a length-indexed vector of type $\text{Vec}\ \mathbb{b}\ n$. T_1 translates Γ to a value type $\Gamma^{\times}$. I implement this translation as a postfix operator $\_^{\times}$ that takes a $\text{Vec}\ \mathbb{b}\ n$ and returns a value type of [implementation] type $\mathbb{b}$:

$$\_^{\times} : \forall\{n : \mathbb{N}\} \to \text{Vec}\ \mathbb{b}\ n \to \mathbb{b}$$

$[\Gamma, x : b] \longleftrightarrow b :: \Gamma$ in my implementation. Names are no longer necessary with de Bruijn indices, since the variable names are implicit in the position within the environment vector.

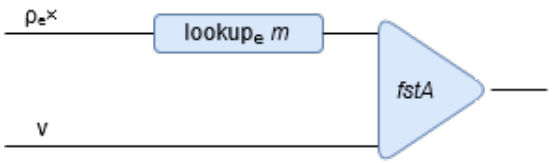
$$([])^{\times} = \mathbb{1} \qquad (\epsilon)_{\mathsf{e}}{}^{\times} = []$$
$$(b :: \Gamma)^{\times} = ((\Gamma)^{\times}) \times b \qquad (\rho +_{\mathsf{e}} v)_{\mathsf{e}}{}^{\times} = [\ (\rho)_{\mathsf{e}}{}^{\times}\ ,\ v\ ]$$

As outlined in section 2.1.5, we translate this to $\Gamma^{\times} \times b$. The empty type environment ϵ which corresponds to the empty vector [], is translated to the primitive type $\mathbb{1}$.

Translations of value and type environments are compared above. Value environments are translated exactly as described in section 2.1.5, using the postfix operator $\__{\mathsf{e}}{}^{\times}$. Dependent types ensure that the translation of a value environment ρ of type Γ is a value of type $\Gamma^{\times}$:

$$\__{\mathsf{e}}{}^{\times} : \forall\{n : \mathbb{N}\} \to \forall\{\Gamma : \mathsf{Vec}\ \mathbb{b}\ n\} \to \Gamma\ \mathsf{env} \to \mathsf{val}\ ((\Gamma)^{\times})$$

T_1 requires an ML_Π combinator to look up values in value environments. The structure of this combinator depends on the length of the environment. Suppose we want to look up the variable $m+1$, which is at binding level $m+1$. If we exit our current binding (at level 0), our required variable will be at binding level m. So looking up variable $m+1$ is the same as looking up variable m in our environment minus the current binding. Consider **there** m to be the same as **suc** $m = m+1$. **here** and **there** are replacements for **zero** and **suc** that verify additional properties when accessing a vector.



$$\mathsf{lookup}_{\mathsf{e}}\ (b :: \Gamma)\ (\mathsf{there}\ m) = (\mathsf{first}\ (\mathsf{lookup}_{\mathsf{e}}\ \Gamma\ m)) \ggg \mathsf{fstA}$$

$\mathsf{lookup}_{\mathsf{e}}$ is a function that produces a combinator to look up a value in a translated environment. This translation may be a pair that encodes a list, as in the recursive case shown above. $\mathsf{lookup}_{\mathsf{e}}\ m$ looks up the required value in the translation of the environment without the current binding, i.e. $\rho_{\mathrm{e}}{}^{\times}$. The current binding $[0 = v]$ is ignored. We then use **fstA** to obtain the result of the recursive call. If we want the value at level 0, we would obtain it from the current binding directly, using **sndA**:

$$\mathsf{lookup}_{\mathsf{e}}\ (b :: \Gamma)\ \mathsf{here} = \mathsf{sndA}$$

The declaration of $\mathsf{lookup}_{\mathsf{e}}$ ensures that the combinator always returns a value of the correct type, i.e. given the translation of environment ρ of type Γ, it will return a value of the type b at position x within Γ.

$$\mathsf{lookup}_{\mathsf{e}} : \forall\{n : \mathbb{N}\}\{x : \mathsf{Fin}\ n\}\{b : \mathbb{b}\} \to (\Gamma : \mathsf{Vec}\ \mathbb{b}\ n) \to (\Gamma\ [\ x\ ]{=}\ b) \to ((\Gamma)^{\times}) \rightsquigarrow b$$

	James & Sabry	**LET**	**ML_Π**
ρ	$[x_1 = v_1, x_2 = v_2, x_3 = v_3]$	$\epsilon +_{\mathrm{e}} v_1 +_{\mathrm{e}} v_2 +_{\mathrm{e}} v_3$	$[[[[], v_1], v_2], v_3]$
Γ	$[x_1 = b_1, x_2 = b_2, x_3 = b_3]$	$b_3 :: b_2 :: b_1 :: \epsilon$	$((1 \times b_1) \times b_2) \times b_3$
Ref	x_2	1	**Shown below**

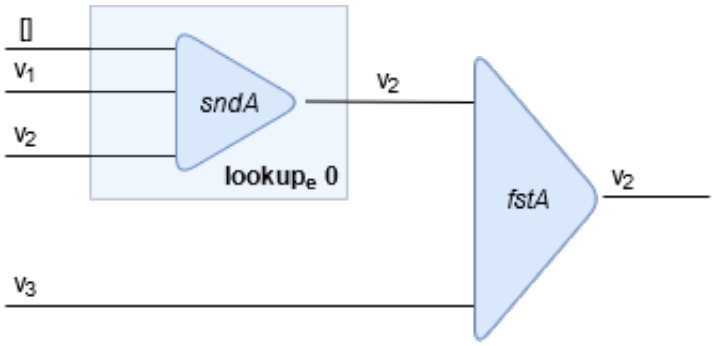


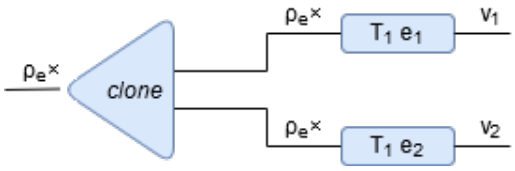
Figure 3.3: Example showing T_1 on environments as well as $\mathrm{lookup_e}\ \Gamma^\times\ 1$ in action

3.3.2 Translating LET constructs into ML_Π combinators

This is at the heart of the implementation of T_1 and also proved to be the least challenging.

$$\mathsf{T}_1 : \forall\{n : \mathbb{N}\} \to \forall\{\Gamma : \mathsf{Vec}\ \mathbb{b}\ n\} \to \forall\{b : \mathbb{b}\} \to \Gamma \vdash \mathsf{exp}:\ b \to ((\Gamma)^\times) \rightsquigarrow b$$

In the declaration of T_1, we specify that given an expression e that has the type b in the type environment Γ, T_1 will return an ML_Π combinator that returns a value of type b when given the translation of an environment of type Γ.



$$\mathsf{T}_1\ \{\Gamma = \gamma\}\ (\langle\ e_1\ ,\ e_2\ \rangle_\mathsf{e}) = (\mathsf{clone}\ ((\gamma)^\times)) \ggg ((\mathsf{T}_1\ e_1) \otimes (\mathsf{T}_1\ e_2))$$

Above I show translation of pairs. We **clone** the environment and run the translations of the subexpressions e_1 and e_2. The type environment Γ is passed as an implicit parameter: Agda can work this out from the type of the expression.

I modified T_1 on **case** from [10] to remove redundant branching, making it more memory-efficient:

$$\begin{aligned}
&\mathsf{T}_1\ \{\Gamma = \gamma\}\ (_\mathsf{e}\mathsf{case}\ e\ _\mathsf{e}\mathsf{L}\ e_1\ _\mathsf{e}\mathsf{R}\ e_2) = \\
&\qquad ((\mathsf{clone}\ ((\gamma)^\times)) \ggg ((\mathsf{first}\ (\mathsf{T}_1\ e)) \ggg ((\mathsf{arr\ distrib}) \ggg \\
&\qquad\qquad ((((\mathsf{arr\ swap}^\times) \ggg (\mathsf{T}_1\ e_1)) \ggg (\mathsf{arr\ uniti})) \oplus \\
&\qquad\qquad\qquad ((((\mathsf{arr\ swap}^\times) \ggg (\mathsf{T}_1\ e_2)) \ggg (\mathsf{arr\ uniti}))))))) \ggg \\
&\qquad\qquad\qquad ((\mathsf{arr\ factor}) \ggg \mathsf{sndA})
\end{aligned}$$

3.3.3 Proof of correctness for T_1: $(e, \rho)_{\mathbf{LET}} = (\mathbf{T}_1\ e)_{\mathbf{ML}_\Pi}\ \rho_\mathbf{e}{}^\times$

Theorem 1. *$(T_1\ e)_{ML_\Pi}\ \rho_e{}^\times = (e, \rho)_{LET}$*

This will be proven using structural induction on e. I will discuss proof formalisation after a pen-and-paper proof for the **case** construct. Here I use the unmodifed definition of T_1. The formalised proof remains unchanged since both definitions have the same normal forms.

Proof. Case **case**: There can be one of 2 evaluation rules applicable here. Assume it is the one below:

$$\frac{(e,\rho) \Downarrow \text{right } v \quad (e_2, [\rho,\, y = v]) \Downarrow v_2}{(\text{case } e \text{ in } \mathrm{L}x \to e_1 \ , \ \mathrm{R}y \to e_2 \ , \rho) \Downarrow v_2}\ \mathbf{case_{eR}} \tag{a}$$

Let the type environment be Γ.

$$\begin{gathered} \mathrm{T}_1\ (\mathbf{case}\ e\ \mathbf{in}\ \mathrm{L}x \to e_1\ ,\ \mathrm{R}y \to e_2) \\ = \\ \mathbf{clone}\ \Gamma^{\times} \ggg \text{first}\ (\mathrm{T}_1\ e) \ggg \text{arr distrib} \ggg \\ (\text{arr swap}^{\times}) \oplus (\text{arr swap}^{\times}) \ggg (\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2) \ggg \mathbf{join} \end{gathered} \tag{b}$$

By induction on the left subderivation in (a),

$$(\mathrm{T}_1\ e)\ {\rho_{\mathrm{e}}}^{\times} = (e,\rho) = \text{right } v \tag{c}$$

By induction on the right subderivation in (a),

$$(\mathrm{T}_1\ e_2)\ [{\rho_{\mathrm{e}}}^{\times},\, v] = (e_2,\, [\rho,\, y = v]) = v_2 \tag{d}$$

Expanding using (b) and the semantics of $\ggg$,

$$\begin{aligned} & \mathrm{T}_1\ (\mathbf{case}\ e\ \mathbf{in}\ \mathrm{L}x \to e_1\ ,\ \mathrm{R}y \to e_2)\ {\rho_{\mathrm{e}}}^{\times} \\ = \ & \mathbf{join}\ (((\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2))\ (((\text{arr swap}^{\times}) \oplus (\text{arr swap}^{\times})) \\ & \quad ((\text{arr distrib})\ ((\text{first}\ (\mathrm{T}_1\ e))\ ((\mathbf{clone}\ \Gamma^{\times})\ {\rho_{\mathrm{e}}}^{\times})))) \\ = \ & \mathbf{join}\ (((\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2))\ (((\text{arr swap}^{\times}) \oplus (\text{arr swap}^{\times})) \\ & \quad ((\text{arr distrib})\ ((\text{first}\ (\mathrm{T}_1\ e))\ ([{\rho_{\mathrm{e}}}^{\times},\, {\rho_{\mathrm{e}}}^{\times}])))) \qquad \text{[Semantics of } \mathbf{clone}\text{]} \\ = \ & \mathbf{join}\ (((\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2))\ (((\text{arr swap}^{\times}) \oplus (\text{arr swap}^{\times})) \\ & \quad ((\text{arr distrib})\ ([\text{right } v,\, {\rho_{\mathrm{e}}}^{\times}]))) \qquad \text{[Using (c) and semantics of } \mathbf{first}\text{]} \\ = \ & \mathbf{join}\ (((\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2))\ (((\text{arr swap}^{\times}) \oplus (\text{arr swap}^{\times})) \\ & \quad (\text{right}\ [v,\, {\rho_{\mathrm{e}}}^{\times}])) \qquad \text{[Semantics of } \mathbf{distrib}\text{]} \\ = \ & \mathbf{join}\ (((\mathrm{T}_1\ e_1) \oplus (\mathrm{T}_1\ e_2))\ (\text{right}\ [{\rho_{\mathrm{e}}}^{\times},\, v]) \qquad \text{[Semantics of } \mathbf{swap}^{\times}\text{]} \\ = \ & \mathbf{join}\ (\text{right}\ v_2) \qquad \text{[Using (d)]} \\ = \ & v_2 \qquad \text{[Semantics of } \mathbf{join}\text{]} \end{aligned} \tag{e}$$

Putting (a) and (e) together, we have:

$$\mathrm{T}_1\ (\mathbf{case}\ e\ \mathbf{in}\ \mathrm{L}x \to e_1\ ,\ \mathrm{R}y \to e_2)\ {\rho_{\mathrm{e}}}^{\times} = v_2 = (\mathbf{case}\ e\ \mathbf{in}\ \mathrm{L}x \to e_1\ ,\ \mathrm{R}y \to e_2,\ \rho)$$

□

Theorem 1 is expressed as T_1-proof, which outputs a proof that $(e,\rho)_{\mathrm{LET}} = (\mathrm{T}_1\ e)_{\mathrm{ML}_\Pi}\ {\rho_{\mathrm{e}}}^{\times}$.

$$\begin{aligned} & \mathrm{T}_1\text{-proof} : \forall\{n : \mathbb{N}\}\{\Gamma : \mathsf{Vec}\ \mathbb{b}\ n\}\{b : \mathbb{b}\} \\ & \qquad \to (\rho : \Gamma\ \mathsf{env}) \to (e : \Gamma \vdash \mathsf{exp}:\ b) \to \mathsf{eval_e}\ \rho\ e \equiv (\mathsf{T}_1\ e)\ [\ ((\rho)_{\mathsf{e}}{}^{\times})\ ]^{\mathsf{a}} \end{aligned}$$

We must provide to the typechecker, our assumption that (e,ρ) evaluates to **right** v for some value v via the **with** construct for pattern matching. The LHS is $\mathbf{eval_e}\ \rho\ \mathbf{(case_e\ e\ {}_eL\ e_1\ {}_eR\ e_2)}$ and Agda cannot reduce it unless it knows the tag of $\mathrm{eval_e}\ \rho\ \mathrm{e}$.

Now Agda reduces the LHS to $\text{eval}_e\ (\rho +_e v)\ e_2$, using (d) to further reduce to $(T_1\ e_2)\ [\rho_e{}^\times, v]$. Unable to do any more LHS simplification, Agda expands the RHS, currently at the first line of (e), to the second line using T_1 and $\ggg$ semantics. We help Agda get to the third line by providing it a proof that **clone** $\Gamma^\times$ on $\rho_e{}^\times$ produces $[\rho_e{}^\times, \rho_e{}^\times]$. To step to the fourth line, we provide a proof that $(T_1\ e)\ \rho_e{}^\times = \text{eval}_e\ \rho\ e =$ **right** v by transitively combining $(T_1\ e)\ \rho_e{}^\times = \text{eval}_e\ \rho\ e$ and $\text{eval}_e\ \rho\ e =$ **right** v, the equality implied by **with** which we obtain via the **inspect** idiom. Via the semantics of $\oplus$, **swap**$^\times$ and **distrib**, Agda reaches the sixth line. All equality proofs are provided via **rewrite**, which allows Agda to replace instances of the LHS with the RHS with any goal it is trying to prove.

Deviating from the proof above, Agda uses **join** to equate the RHS to $\mathbf{(T_1\ e_2)\ [\rho_e{}^\times, v]}$, which is the simplified LHS. The proof is complete:

```
T₁-proof ρ (ₑcase e ₑL e₁ ₑR e₂) with (evalₑ ρ e) | inspect (evalₑ ρ) e
... | right v | [ pf ]
  rewrite (clone-proof ((ρ)ₑ^×)) | (T₁-proof (ρ +ₑ v) e₂)
                                 | (trans (sym (T₁-proof ρ e)) pf) = refl
```

Lemma 1. *For any environment ρ of type Γ, $\rho[x] = (lookup_e\ \Gamma\ x)_{ML_\Pi}\ \rho_e{}^\times$ - Proof in Appendix A.*

3.4 Π - the low-level reversible language

3.4.1 Formalisation of syntax and operational semantics

As for ML_Π, Π combinators are also indexed by their input and output types for type-safety guarantees. The ∀ in the representative constructors for primitive constructs is because, for instance, **assocl**$^+$ is of type $b_1 + (b_2 + b_3) \to (b_1 + b_2) + b_3$ for any b_1, b_2 and b_3.

```
data _↔_ : 𝕓 → 𝕓 → Set where

assocl₊ : ∀{b₁ b₂ b₃} → (b₁ + (b₂ + b₃)) ↔ ((b₁ + b₂) + b₃)
assocr₊ : ∀{b₁ b₂ b₃} → ((b₁ + b₂) + b₃) ↔ (b₁ + (b₂ + b₃))
```

Like in LET, typing rules has been incorporated into constructors for composite constructs:

```
_×_ : ∀{b₁ b₂ b₃ b₄}
      → b₁ ↔ b₂ → b₃ ↔ b₄
      --------------------
      → (b₁ × b₃) ↔ (b₂ × b₄)
```

The interpreter for evaluation was straightforward except for **sym** c (**symm** c in my implementation) which is functionally the same as $c^\dagger$. An interpreter $\_[\_]^f$ that evaluates combinator c on a value v might be defined for this case as: $(\text{symm}\ c)\ [v]^f = (c^\dagger)\ [v]^f$

Unfortunately, the above will fail Agda's termination checking, which cannot prove that **symm** c will terminate when evaluated: it cannot prove that $c^\dagger$ is structurally smaller than **symm** c. Termination for recursive functions can only be proved if the recursive call is on structurally smaller arguments. So now I define two mutually recursive interpreters $\_[\_]^f$ and $\_[\_]^b$ for forward and backward evaluation respectively:

$$\_[\_]^{\mathsf{f}} : \forall \{b\ b'\} \to b \leftrightarrow b' \to \mathsf{val}\ b \to \mathsf{val}\ b'$$
$$\_[\_]^{\mathsf{b}} : \forall \{b\ b'\} \to b \leftrightarrow b' \to \mathsf{val}\ b' \to \mathsf{val}\ b$$

Running **symm** c forwards on a value v is the same as running c backwards on v:

$$(\mathsf{symm}\ c)\ [\ v\ ]^{\mathsf{f}} = c\ [\ v\ ]^{\mathsf{b}} \qquad\qquad (\mathsf{symm}\ c)\ [\ v\ ]^{\mathsf{b}} = c\ [\ v\ ]^{\mathsf{f}}$$

Similarly to the LET interpreter, we get implicit type-safety proofs in the definitions of the interpreters. For example, $\_[\_]^{\mathsf{f}}$ guarantees that the forward evaluation of a combinator on a value of its input type b will terminate with a value of its return type b'.

Some representative examples of the forward evaluation are shown below. For a sequential composition of c_1 followed by c_2, we evaluate c_1 on a value v first, followed by an evaluation of c_2 on the resultant value. Backward evaluation is derived in the following section.

$$\mathsf{swap}^{\times}\ [\ ([\ v_1\ ,\ v_2\ ])\ ]^{\mathsf{f}} = [\ v_2\ ,\ v_1\ ]$$
$$\mathsf{assocl}^{+}\ [\ \mathsf{left}\ v_1\ ]^{\mathsf{f}} = \mathsf{left}\ (\mathsf{left}\ v_1)$$
$$(c_1 + c_2)\ [\ \mathsf{left}\ v_1\ ]^{\mathsf{f}} = \mathsf{left}\ (c_1\ [\ v_1\ ]^{\mathsf{f}})$$
$$(c_1 + c_2)\ [\ \mathsf{right}\ v_2\ ]^{\mathsf{f}} = \mathsf{right}\ (c_2\ [\ v_2\ ]^{\mathsf{f}})$$
$$(c_1 \times c_2)\ [\ ([\ v_1\ ,\ v_2\ ])\ ]^{\mathsf{f}} = [\ (c_1\ [\ v_1\ ]^{\mathsf{f}})\ ,\ (c_2\ [\ v_2\ ]^{\mathsf{f}})\ ]$$
$$(c_1 > c_2)\ [\ v\ ]^{\mathsf{f}} = c_2\ [\ (c_1\ [\ v\ ]^{\mathsf{f}})\ ]^{\mathsf{f}}$$

3.4.2 An algorithm to compute the inverse $c^\dagger$ of every program construct c

This section derives the adjoints of composite combinators, starting with $c_1 \times c_2$. Adjoints for primitive operators are given in section 2.1.3. The composition of the morphisms in the diagram below gives the identity, which is important because we need a combinator $(c_1 \times c_2)^\dagger$ that, when composed with $c_1 \times c_2$, gives the identity. Because f and g are isomorphisms, $\mathrm{f} \circ \mathrm{f}^{-1} = \mathrm{id}_{B_3}$, $\mathrm{f}^{-1} \circ \mathrm{f} = \mathrm{id}_{B_1}$, $\mathrm{g} \circ \mathrm{g}^{-1} = \mathrm{id}_{B_4}$ and $\mathrm{g}^{-1} \circ \mathrm{g} = \mathrm{id}_{B_2}$. To see why the following morphisms when composed give the identity requires more reasoning.

$$B_1 \times B_2 \underset{\mathrm{f}^{-1}\times\mathrm{g}^{-1}}{\overset{\mathrm{f}\times\mathrm{g}}{\rightleftarrows}} B_3 \times B_4$$

Reasoning for the forwards and backwards directions in the product diagram:

	$(\mathrm{f}^{-1} \times \mathrm{g}^{-1}) \circ (\mathrm{f} \times \mathrm{g})$			$(\mathrm{f} \times \mathrm{g}) \circ (\mathrm{f}^{-1} \times \mathrm{g}^{-1})$
$=$	$(\mathrm{f}^{-1} \circ \mathrm{f}) \times (\mathrm{g}^{-1} \circ \mathrm{g})$	[Interchange product and composition]	$=$	$(\mathrm{f} \circ \mathrm{f}^{-1}) \times (\mathrm{g} \circ \mathrm{g}^{-1})$
$=$	$\mathrm{id}_{B_1} \times \mathrm{id}_{B_2}$	[f and g are isomorphisms]	$=$	$\mathrm{id}_{B_3} \times \mathrm{id}_{B_4}$
$=$	$\mathrm{id}_{B_1 \times B_2}$	[Def of identity for products]	$=$	$\mathrm{id}_{B_3 \times B_4}$

Drawing an analogy with Π combinators, consider the typing rule for $c_1 \times c_2$:

$$\frac{c_1 : b_1 \leftrightarrow b_2 \qquad c_2 : b_3 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4}\ \times$$

The types b_1, b_2, b_3 and b_4 are analogous to the category-theoretic objects B_1, B_2, B_3 and B_4 respectively. c_1 is a reversible function from b_1 to b_2. Let it be analogous to the isomorphism

$\mathbf{f}{:}B_1 \to B_2$. Then $c_1{}^\dagger$ that reverses the effect of c_1, should be analogous to $\mathbf{f}^{-1}{:}B_2 \to B_1$. Similarly, let c_2 and $c_2{}^\dagger$ be analogous to g and g^{-1} respectively. Then $c_1 \times c_2$, from $b_1 \times b_2$ to $b_3 \times b_4$, corresponds to $\mathbf{f} \times \mathbf{g}$: $B_1 \times B_2 \to B_3 \times B_4$.

We know that the inverse of $\mathbf{f} \times \mathbf{g}$ is $\mathbf{f}^{-1} \times \mathbf{g}^{-1}$. From here we conjecture that since $c_1 \times c_2$ corresponds to $\mathbf{f} \times \mathbf{g}$, its inverse $(c_1 \times c_2)^\dagger$ should correspond to $\mathbf{f}^{-1} \times \mathbf{g}^{-1}$, whose interpretation in Π is $c_1{}^\dagger \times c_2{}^\dagger$. So we augment the typing rule for products as follows, where we write the adjoint beside the type of the combinators:

$$\frac{c_1 : b_1 \leftrightarrow b_2 : c_1{}^\dagger \qquad c_2 : b_3 \leftrightarrow b_4 : c_2{}^\dagger}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4 : c_1{}^\dagger \times c_2{}^\dagger}\ \times$$

Now we consider the backwards evaluation in the product case, but let us first have a more general discussion. Consider a combinator c of type $b \leftrightarrow b'$. Running it 'backwards' would mean giving it a value v' of type b' and then evaluating it in reverse, which is the same as evaluating its inverse $c^\dagger$ on v' in the forward direction. But we cannot write the backwards evaluation in the form $\mathrm{c[v]^b = c^\dagger[v]^f}$. If we could, then for the product case, we would have the following definition in our implementation:

$$(c_1 \times c_2)[[v_1,\, v_2]]^\mathrm{b} = (c_1{}^\dagger \times c_2{}^\dagger)[[v_1,\, v_2]]^\mathrm{f}$$

Without any further information about c_1 and c_2, Agda's termination checking cannot show that $c_1{}^\dagger \times c_2{}^\dagger$ is structurally smaller than $c_1 \times c_2$ and hence cannot guarantee that the recursion will terminate. Therefore we must express the RHS in such a way that any arguments to $\_[\_]^\mathrm{f}$ and $\_[\_]^\mathrm{b}$ are structurally smaller than $c_1 \times c_2$. Noting the equality $\mathrm{c[v]^b = c^\dagger[v]^f}$ and recalling the forward evaluation rule for products:

$$\frac{c_1\ v_1 \mapsto v_3 \qquad c_2\ v_2 \mapsto v_4}{c_1 \times c_2\ [v_1{,}v_2] \mapsto [v_3{,}v_4]}\ \mathrm{eval}_\times$$

We can now write: $(c_1 \times c_2)[[v_1,\, v_2]]^\mathrm{b} = (c_1{}^\dagger \times c_2{}^\dagger)[[v_1,\, v_2]]^\mathrm{f} = [c_1{}^\dagger[v_1]^\mathrm{f},\, c_2{}^\dagger[v_2]^\mathrm{f}] = [c_1[v_1]^\mathrm{b},\, c_2[v_2]^\mathrm{b}]$. The backwards evaluation rule for products is thus:

$$\frac{v_3 \mathrel{\text{↤}} c_1\ v_1 \qquad v_4 \mathrel{\text{↤}} c_2\ v_2}{[v_3{,}v_4] \mathrel{\text{↤}} (c_1 \times c_2)\ [v_1{,}v_2]}\ \mathrm{eval}^\mathrm{b}{}_\times$$

The reasoning for sums is similar to that for products, and so the augmented typing rule with the adjoint is:

$$\frac{c_1 : b_1 \leftrightarrow b_2 : c_1{}^\dagger \qquad c_2 : b_3 \leftrightarrow b_4 : c_2{}^\dagger}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4 : c_1{}^\dagger + c_2{}^\dagger}\ +$$

For backwards evaluation, $(c_1 + c_2)[\mathrm{left}\ v]^\mathrm{b} = (c_1{}^\dagger + c_2{}^\dagger)[\mathrm{left}\ v]^\mathrm{f} = \mathrm{left}\ (c_1{}^\dagger[v]^\mathrm{f}) = \mathrm{left}\ (c_1[v_1]^\mathrm{b})$. Similarly, $(c_1 + c_2)[\mathrm{right}\ v]^\mathrm{b} = \mathrm{right}\ (c_2[v]^\mathrm{b})$.

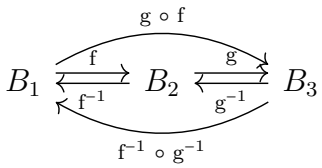
$$\frac{v_3 \mathrel{\text{↤}} c_1\ v_1}{\mathrm{left}\ v_3 \mathrel{\text{↤}} (c_1 + c_2)\ \mathrm{left}\ v_1}\ \mathrm{eval}^\mathrm{b}{}_{\mathrm{L}_+} \qquad \frac{v_4 \mathrel{\text{↤}} c_2\ v_2}{\mathrm{right}\ v_4 \mathrel{\text{↤}} (c_1 + c_2)\ \mathrm{right}\ v_2}\ \mathrm{eval}^\mathrm{b}{}_{\mathrm{R}_+}$$

sym c is the inverse of c, so its reverse $(\mathbf{sym}\ c)^\dagger$ is simply c, and running it backwards on a value v is the same as running c on v in the opposite direction. The reverse of the identity is simply the identity.

$$\frac{c : b_1 \leftrightarrow b_2 : c^\dagger}{\mathrm{sym}\ c : b_2 \leftrightarrow b_1 : c}\ \mathrm{sym} \qquad \frac{c\ v_1 \mapsto v_2}{v_2 \mathrel{\text{↤}} (\mathrm{sym}\ c)\ v_1}\ \mathrm{eval}^\mathrm{b}{}_\mathrm{sym}$$

$$\frac{}{\mathrm{id} : b \leftrightarrow b : \mathrm{id}}\ \mathrm{id} \qquad \frac{}{v \mathrel{\text{↤}} \mathrm{id}\ v}\ \mathrm{eval}^\mathrm{b}{}_\mathrm{id}$$

Finally, for the sequential combinators, we have the following category-theoretic diagram:



Let c_1, $c_1{}^\dagger$, c_2 and $c_2{}^\dagger$ correspond to f, f^{-1}, g and g^{-1} respectively. Hence $c_1;c_2$, the sequential composition of c_1 and c_2, is analogous to $\mathbf{g} \circ \mathbf{f}$, whose inverse is $\mathbf{f}^{-1} \circ \mathbf{g}^{-1}$. $(c_1;c_2)^\dagger$ must be represented by the latter, whose reprentation in Π is $c_2{}^\dagger;c_1{}^\dagger$:

$$\frac{c_1 : b_1 \leftrightarrow b_2 : c_1{}^\dagger \qquad c_2 : b_2 \leftrightarrow b_3 : c_2{}^\dagger}{c_1;c_2 : b_1 \leftrightarrow b_3 : c_2{}^\dagger;c_1{}^\dagger} \ ;$$

f and g are isomorphisms, so $f \circ f^{-1} = id_{B_2}$, $f^{-1} \circ f = id_{B_1}$, $g \circ g^{-1} = id_{B_3}$ and $g^{-1} \circ g = id_{B_2}$. So we have the following proofs:

$(f^{-1} \circ g^{-1}) \circ (g \circ f) = f^{-1} \circ (g^{-1} \circ g) \circ f = f^{-1} \circ id_{B_2} \circ f = f^{-1} \circ f = id_{B_1}$

$(g \circ f) \circ (f^{-1} \circ g^{-1}) = g \circ (f \circ f^{-1}) \circ g^{-1} = g \circ id_{B_2} \circ g^{-1} = g \circ g^{-1} = id_{B_3}$

$(c_1;c_2)[v]^b = (c_2{}^\dagger;c_1{}^\dagger)[v]^f = c_1{}^\dagger[c_2{}^\dagger[v]^f]^f = c_1[c_2[v]^b]^b$, giving us the evaluation rule:

$$\frac{v_3 \leftarrow c_1\ v_2 \qquad v_2 \leftarrow c_2\ v_1}{v_3 \leftarrow v_1\ c_1;c_2}\ \text{eval}^b;$$

3.4.3 Proof of Π's reversibility, i.e. for every c, if $c\ v \mapsto v'$, then $c^\dagger\ v' \mapsto v$

We want to prove that for any combinator c in Π, if $c\ v \mapsto v'$, then $c^\dagger\ v' \mapsto v$. We know that Π is strongly normalising, which means that if v is a value of c's input type, then it always holds that $c\ v \mapsto v'$ for some value v' of c's output type. Thus the assumption $c\ v \mapsto v'$ is always true. So we can replace v' with $c\ v$ in $c^\dagger\ v' \mapsto v$ and prove the following theorem instead :

Theorem 2. *For any combinator c: $b \leftrightarrow b'$ and a value v of type b, it holds that $c^\dagger(c\ v) \mapsto v$.*

$\mapsto$ corresponds to forward evaluation, i.e. $\_[\_]^f$. This theorem has been formalised as follows:

```
Π-rev-proof : ∀ {b b'}(c : b ↔ b')(v : val b) → (c †) [ c [ v ]f ]f ≡ v
```

The proof for the primitive isomorphisms and **id** is trivial, which is evident in the **refl** on the RHS of the corresponding patterns. Consider as a representative example **distrib** : $(b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3)$. Its input must be of the form [**left** v_1, v_3] or [**right** v_2, v_3]. From the operational semantics we have that:

$$\textbf{distrib}\ [\mathit{left}\ v_1{,}v_3] \mapsto \mathit{left}\ [v_1{,}v_3]$$
$$\textbf{distrib}\ [\mathit{right}\ v_2{,}v_3] \mapsto \mathit{right}\ [v_2{,}v_3]$$

The adjoint of **distrib**, i.e. **distrib**† is **factor**, whose operational semantics are as follows:

$$\textbf{factor}\ \mathit{left}\ [v_1{,}v_3] \mapsto [\mathit{left}\ v_1{,}v_3]$$
$$\textbf{factor}\ \mathit{right}\ [v_1{,}v_3] \mapsto [\mathit{right}\ v_1{,}v_3]$$

Clearly, **factor** (**distrib** [*left* v_1,v_3]) ↦ [*left* v_1,v_3] and **factor** (**distrib** [*right* v_2,v_3]) ↦ [*right* v_2,v_3]. The typechecker can work this out itself using the definition of adjoints and forward evaluation i.e. using the definitions of _[_]^f and _†, so we do not need to provide any extra equalities in the case when c = **distrib**:

```
Π-rev-proof distrib ([ left v₁ , v₃ ]) = refl
Π-rev-proof distrib ([ right v₂ , v₃ ]) = refl
```

We use structural induction on c for the cases where it is a composition combinator:

Proof. Case ($c = c_1 + c_2$): Below is the proof for when the input value is tagged with **left**. The proof for **right** is similar. We replace ↦ with =.

$$
\begin{aligned}
& (c_1 + c_2)^\dagger\ ((c_1 + c_2)\ \mathbf{left}\ v) && \\
= \quad & (c_1 + c_2)^\dagger\ \mathbf{left}\ (c_1\ v) && \text{[From operational semantics on sums]} \\
= \quad & (c_1{}^\dagger + c_2{}^\dagger)\ \mathbf{left}\ (c_1\ v) && \text{[Defn of adjoints: } (c_1 + c_2)^\dagger = c_1{}^\dagger + c_2{}^\dagger\text{]} \\
= \quad & \mathbf{left}\ (c_1{}^\dagger\ (c_1\ v)) && \text{[From operational semantics on sums]} \\
= \quad & \mathbf{left}\ v && \text{[By induction on } c_1\text{]}
\end{aligned}
$$

Hence shown that the output value of this computation is equal to the input value. For **right**, we do the induction on c_2. □

This proof was the first one to be implemented so there was some experimentation with proof techniques and different cases are proved in different ways. I will evaluate these proof techniques later. The typechecker can get from the first line to the fourth using its knowledge of the semantics and adjoints. It has no knowledge about the structure of c_1, so it cannot simplify **left** ($c_1{}^\dagger$ (c_1 v)) further. So we provide it the equality **left** ($c_1{}^\dagger$ (c_1 v)) = **left** v in the RHS like so:

```
Π-rev-proof (c₁ + c₂) (left v) = cong left (Π-rev-proof c₁ v)
```

cong f eq where eq is an equality of the form $a = b$ and where **f** is a function, outputs an equality of the form **f** a = **f** b. In the above code, eq is the induction hypothesis $c_1{}^\dagger$ (c_1 v) = v represented by **Π-rev-proof** c_1 v and **f** is the constructor **left**, so the RHS is the required equality. We did not provide this equality via **rewrite** because it can be written in the form of our goal, that is, $(c_1 + c_2)^\dagger$ ($(c_1 + c_2)$ **left** v) = **left** v. The equalities provided in T_1–proof via **rewrite** could not be written in the form of our goal: they were merely used to rewrite parts of the RHS or LHS.

We now move to the sequential construct c_1;c_2. As described above, the implementation uses > instead of ;, but other than that, the following is a formalised 'pen-and-paper' style proof for this case:

```
Π-rev-proof (c₁ > c₂) v = begin
                            (c₁ > c₂)† [ (c₁ > c₂) [ v ]ᶠ ]ᶠ      - (a)
                          ≡⟨⟩
                        - Expanding (c₁ > c₂)† using definition of _†
                            ((c₂ †) > (c₁ †)) [ (c₁ > c₂) [ v ]ᶠ ]ᶠ - (b)
                          ≡⟨⟩
```

```
- Expanding (c₁ > c₂) [ v ]ᶠ using definition of _[_]ᶠ
    ((c₂ †) > (c₁ †)) [ c₂ [ c₁ [ v ]ᶠ ]ᶠ ]ᶠ  - (c)
  ≡⟨⟩
- Again using _[_]ᶠ to expand (c)
    (c₁ †) [ (c₂ †) [ c₂ [ c₁ [ v ]ᶠ ]ᶠ ]ᶠ ]ᶠ   - (d)
  ≡⟨ cong ((c₁ †) [_]ᶠ) (Π-rev-proof c₂ (c₁ [ v ]ᶠ)) ⟩
- Induction on c₂ and c₁ [ v ]ᶠ
    (c₁ †) [ c₁ [ v ]ᶠ ]ᶠ - (e)
  ≡⟨ Π-rev-proof c₁ v ⟩
- Induction on c₁ and v
  v - (f)
  ■
```

In between expressions which are equal to each other, we must provide proofs that they are equal if the typechecker cannot equate them through normalisation. For example, stepping from (d) to (e) requires knowledge that $c_2^{\dagger}\ (c_2\ (c_1\ v)) = c_1\ v$. This is the induction hypothesis that we get via induction on c_2 and $c_1\ v$, so we must provide this to the typechecker through Π**-rev-proof** $c_2\ (c_1\ v)$.

The product case $c = c_1 \times c_2$ and $v = [v_1, v_2]$ can be proved using induction on (c_1, v_1) and (c_2, v_2). It has been formalised as follows and the relevant induction hypotheses have been provided. The formalisation will not be discussed further here since I will make a wider point about congruence in my evaluation of proof techniques:

```
Π-rev-proof (c₁ × c₂) ([ v₁ , v₂ ]) = []-cong (Π-rev-proof c₁ v₁) (Π-rev-proof c₂ v₂)
```

Once again, **sym** causes problems. This is because the forward evaluation of **sym** c has been defined as the backward evaluation of c. The problem crops up in both the implementation and our informal proof.

$$\frac{v_2 \leftarrowtail c\ v_1}{(\text{sym}\ c)\ v_1 \mapsto v_2}\ \text{eval}_{\text{sym}}$$

$(\textbf{sym}\ c)^{\dagger}\ ((\textbf{sym}\ c)\ v) = c\ ((\textbf{sym}\ c)\ v)$
We need to show that $c\ ((\textbf{sym}\ c)\ v) \mapsto v$, i.e. $c\ v' \mapsto v$ where $(\textbf{sym}\ c)\ v \mapsto v' \Leftrightarrow v' \leftarrowtail c\ v$. This can be expressed as the following lemma:

Lemma 2. *For any combinator c: $b \leftrightarrow b'$ and values v of type b and v' of type b', if $v \leftarrowtail c\ v'$ then $c\ v \mapsto v'$.*

All cases other than **sym** can be proved by structural induction on c. For **sym**, we need to prove **Lemma 2** by mutual induction with **Lemma 3**:

Lemma 3. *For any combinator c: $b \leftrightarrow b'$ and values v of type b and v' of type b', if $c\ v \mapsto v'$ then $v \leftarrowtail c\ v'$.*

Note that for **Lemma 2**, we can write $v = c[v']^{\text{b}}$ as v will exist uniquely. Thus, an equivalent statement we can prove is $c[\ c[v']^{\text{b}}\ ]^{\text{f}} = v'$. Similarly, an equivalent statement for **Lemma 3** is $c[\ c[v]^{\text{f}}\ ]^{\text{b}} = v$. The lemmas have been formalised as follows:

```
lemma-2 : ∀ {b b'} → ∀(c : b ↔ b') → ∀ (v' : val b') → c [ c [ v' ]ᵇ ]ᶠ ≡ v'
lemma-3 : ∀ {b b'} → ∀(c : b ↔ b') → ∀ (v : val b) → c [ c [ v ]ᶠ ]ᵇ ≡ v
```

Proof. Showing the **sym** case for **Lemma 2**. So $c = \textbf{sym}\ c'$. Assume:

$$v \nleftarrow (\textbf{sym}\ c')\ v' \tag{a}$$

From $\text{eval}^{\text{b}}_{\text{sym}}$, (a) implies:

$$c'\ v' \mapsto v \tag{b}$$

From **Lemma 3** by induction on c' and v':

$$c'\ v' \mapsto v \implies v' \nleftarrow c'\ v \tag{c}$$

From (b) and (c):

$$v' \nleftarrow c'\ v \tag{d}$$

From eval_{sym} and (d):

$$(\textbf{sym}\ c')\ v \mapsto v' \tag{e}$$

sym for **Lemma 3** can be proved similarly using induction with **Lemma 2**. All other cases for **Lemma 3** are either trivial or by induction with **Lemma 3**. □

My supervisor Prof. Griffin helped me deal with the **sym** case by advising me to implement backward evaluation and prove reversibility with mutually recursive **Lemma 2** and **Lemma 3**.

3.4.4 From Π to Π°: Making Π Turing-complete

Recursive types

We must extend Π with natural numbers and looping computations. In ML, we would write $\mathbb{N} = \textbf{zero} \mid \textbf{succ of}\ \mathbb{N}$. This is an example of a recursive type, i.e. a type that refers to itself in its definition. A mechanism for doing so would be via free variables in types. As before, I use de Bruijn indices as variables and so I index the type of [Π°] types with the number of free variables in the type (n) to allow de Bruijn indices to be values of Fin n. In [10], the set of finite types is extended with the recursive type $\mu x.b$ where b refers to a type and x is the variable. In my implementation [the x must always be 0]:

```
μ_ : ∀{n : ℕ}
     → 𝕓 (suc n)
     ------
     → 𝕓 n
```

b must have one more free variable (x) than $\mu x.b$, hence it is indexed by **suc** n. Similar to the ML definition, we could define $\mathbb{N} = \mathbb{1}$ **or (succ of** x**)**, where x refers to $\mathbb{N}$ itself. Encoding the choice of constructor as a sum type, we get $\mathbb{N} = \mu x.\mathbb{1} + x$. It is easy to see how we can obtain a value of type $\mathbb{1} + \mathbb{N}$, which is essentially $(\mathbb{1} + x)[\mathbb{N}/x]$, but once we have that, how do we create a value of type $\mathbb{N}$?

[10] defines a combinator **fold** for doing this. **fold** : $\mu x.b \rightleftharpoons b[\mu x.b/x]$: **unfold**. These are inverses of each other, which is easily seen from their semantics: **fold** : $v \leftrightarrow \langle v \rangle$: **unfold**. **unfold** 'opens' up a recursive type to allow us to look inside. For instance, by **unfolding** a value of $\mathbb{N}$, we can see whether it is **zero** or the **successor** of something. So, for example, 2 = **succ(succ zero)** in ML, becomes ⟨ **right** ⟨ **right** ⟨ **left** () ⟩ ⟩ ⟩. Π° uses ⇌ for combinators instead of ↔ because its combinators are partial, not total, bijections.

I use an existing Agda implementation of recursive types [11] which allows type substitution for the largest reference in a type. While this works for natural numbers, it does not allow the usage of a type that has n free variables where we need a type that has at most $n+1$ free variables. We need a notion akin to subtyping here.

Looping computations

These are expressed using the **trace** composite combinator in [10]:

$$\frac{c : b_1 + b_2 \rightleftharpoons b_1 + b_3}{\mathbf{trace}\ c : b_2 \rightleftharpoons b_3}\ \text{trace}$$

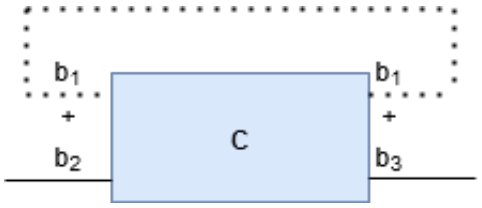


Figure 3.4: Trace c

The idea behind **trace** is that it takes a value v of type b_2, tags it with **right** (so the value becomes **right** v), then runs c on this value continually until we again obtain a value tagged with **right**. It then extracts this tag, and we are left with a value of type b_3. The computation may never terminate if we always get values tagged with **left**. This combinator was challenging to formalise, because [10] expresses in the semantics, the looping version of c that loops until we see a value tagged with **right**, as a construct $\mathbf{loop}_c$. However, in my formalisation, I had to define a function that makes c loop, since $\mathbf{loop}_c$ cannot actually be a part of the formal semantics, given that we cannot define its inverse easily in terms of c:

$$\mathsf{loop} : \{n : \mathbb{N}\}\{b_1\ b_2\ b_3 : \mathbb{b}\ n\} \to (b_1 + b_2) \rightleftharpoons (b_1 + b_3) \to \mathsf{val}\ n\ (b_1 + b_3) \to \mathsf{val}\ n\ b_3$$

[10] states that the adjoint of **trace** c is **trace** $c^\dagger$. Formalising the proof that they are indeed inverses of each other proved to be difficult, so I postulated this in my implementation. The proof is by induction on the number of times we run c while looping, and the objective of the proof is to show that if we have a value **right** v_1 and we have to run c n times until we halt at a value **right** v_{n+1}, then $c^\dagger$ will also loop n times starting at **right** v_{n+1} and halt at **right** v_1. It turns out we must prove something a little more general. Below, the tag of v_{n+1} does not matter, so we leave it out:

Lemma 4. $\forall n.\ \mathit{right}\ v_1 \xrightarrow{c}_n v_{n+1} \implies v_{n+1} \xrightarrow{c^\dagger}_n \mathit{right}\ v_1.$

The first observation is that all intermediate values between **right** v_1 and v_{n+1} are tagged with **left** since we are looping. The base case $n = 1$ follows trivially from **Theorem 2**.

Inductive case: assume Lemma for n. Assume right $v_1 \xrightarrow{c}_{n+1} v_{n+2}$. Then there must exist a v_{n+1} such that right $v_1 \xrightarrow{c}_n v_{n+1}$ and $v_{n+1} \xrightarrow{c} v_{n+2}$. Since v_{n+1} is tagged with **left**, looping will continue and by **Theorem 2**, we have $v_{n+2} \xrightarrow{c^\dagger} v_{n+1}$. By induction, $v_{n+1} \xrightarrow{c^\dagger}_n$ right v_1. So by transitivity of computations, we have $v_{n+2} \xrightarrow{c^\dagger}_{n+1}$ right v_1. ■

3.5 T_2 - the translation from ML_Π to Π

Before talking about the specifics of the implementation of T_2, I will continue the discussion of **left** a from section 2.1.6. Let us consider the evaluation of **left** a, where a: $b_1 \rightsquigarrow b_2$. It takes

in a value v of sum type $b_1 + b_3$. If v has a **left** tag, then **left** a applies a to it. Otherwise, v passes through unchanged. So a first attempt at building the heap and garbage may follow the reasoning as described below.

Suppose a's heap type is h. Then we definitely require $\phi(h)$, the default value of type h in case v has a **left** tag since we will do a computation of a. This computation will also produce a garbage value v_g, where g is a's garbage type. In case v is tagged with **right**, we will not use h, and so $\phi(h)$ will be in the garbage. And in case v is tagged with **left**, v_g will be in the garbage. So it seems reasonable to have the heap type of this computation be h and the garbage type be $h + g$, which would give us $\mathbf{T}_2$ **(left** a**)**: $h \times (b_1 + b_3) \leftrightarrow (h + g) \times (b_2 + b_3)$.

Unfortunately, the above does not work. This is because we are using a single tag in the input v to decide two tags in the output: if v has a **right** tag, then the garbage is tagged with **left** and the relevant output (of type $b_2 + b_3$) with **right**. This is **duplication of branching information** and is not expressible in Π. We could attempt to solve this problem by adding a default value **left ()** to the heap, and use this tag on each of the branches to put a tag on the garbage produced in each case. This fails conceptually, because both branches get the same **left** tag, and neither has any information about the other branch's garbage type. So we cannot unify the results from the branches as both the relevant values and the garbage produced are of different types.

The last failure is actually quite informative. What if the branches determine the type of the garbage but both produce output values of the same type? This idea also works if the garbage produced is of the same type but we do not know the required type of the garbage yet. However, we do know that $(b_2 + b_3)$ is the required output type. So what values do we need to produce this output type in each case?

- v has a **left** tag: We need the default value $\phi(h)$ for the computation of a. Once we have a value v_2 of type b_2, we need to put a **left** tag on it. We have already used up v's tag in deciding to take this branch, but the **leftSwap** combinator can, given values v_1 and **left** v_2 of type b_2 and $(b_2 + b_3)$ produce a value **left** v_1 of type $b_2 + b_3$. So this branch will need a default value **left** $\phi(b_2)$ of type $b_2 + b_3$.

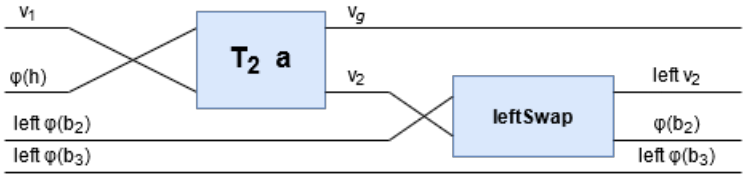


Figure 3.5: **left** branch for T_2 (**left** a)

- v has a **right** tag: $v =$ **right** v_2, where v_2 is of type b_3. To produce a value **right** v_2, we can first use **leftSwap** on v_2 and the default value **left** $\phi(b_3)$ of type $b_3 + b_2$ to get **left** v_2 and then run **swap**$^+$ to change the tag to **right**.

We must provide all default values that may be required by either of the branches in the heap, since we do not yet know which branch will be taken. Any default values not yet used will become garbage for that branch. Our set of all possible heap values is $\phi(h)$, **left** $\phi(b_2)$ and **left** $\phi(b_3)$, so our heap can have type $h \times ((b_2 + b_3) \times (b_3 + b_2))$. Call this type $\mathbf{H}$.

Each branch puts the heap values it does not need into the garbage. The left branch does not require the defualt value of type $b_3 + b_2$. It also produces garbage of type g from

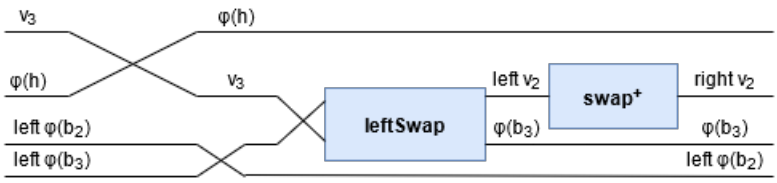
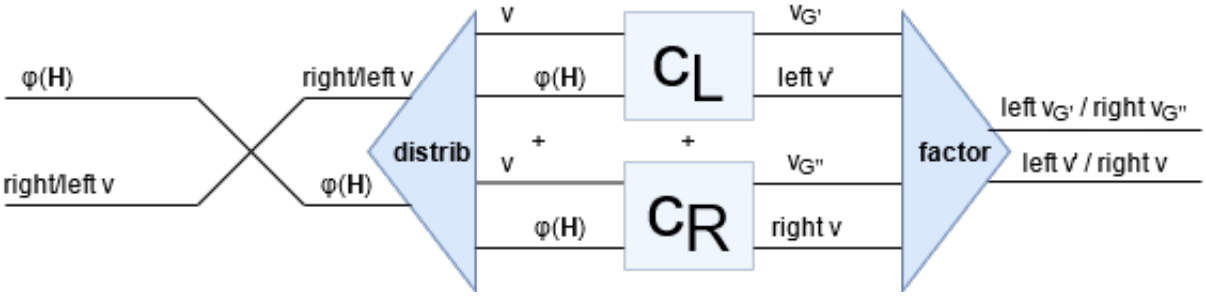


Figure 3.6: **right** branch for T_2 (**left** a)

the computation of a and the default value $\phi(b_2)$ from extracting the tag of $\phi(b_2 + b_3)$ using **leftSwap**. So the garbage from the left branch is of type $g \times (b_2 \times (b_3 + b_2))$ (call this type **G'**). Similarly, the right branch does not need either $\phi(h)$ or $\phi(b_2 + b_3)$. It also produces a garbage value of $\phi(b_3)$ from extracting the tag of $\phi(b_3 + b_2)$ using **leftSwap**. So the right branch produces garbage of type $(h \times (b_2 + b_3)) \times b_3$ (call this type **G"**). I call the combinators for the left and right branches c_L and c_R respectively.



$\mathbf{T_2}$ **(left** a**)** = **swap**$^+$; **distrib**; $c_L + c_R$; **factor**

Figure 3.7: T_2 (**left** a)

Derivation of heap and garbage types for $a_1 \oplus a_2$ is discussed in the context of T_2's implementation. For every combinator a: $b_1 \rightsquigarrow b_2$ in ML_Π, we need to produce a combinator ($\mathbf{T_2}$ a): a: $h \times b_1 \rightsquigarrow g \times b_2$ in Π, where h and g are a's heap and garbage types. I check that the translated combinator has the correct type in my declaration of T_2:

$$T_2 : \forall\{b_1\ b_2\} \to (c : (b_1 \rightsquigarrow b_2)) \to (\mathsf{heap}(c) \times b_1) \leftrightarrow (\mathsf{garbage}(c) \times b_2)$$

heap and garbage are mutually recursive functions that define the heap and garbage types of an ML_Π combinator:

$$\mathsf{heap} : \forall\{b\ b'\} \to b \rightsquigarrow b' \to \mathbb{b}$$
$$\mathsf{garbage} : \forall\{b\ b'\} \to b \rightsquigarrow b' \to \mathbb{b}$$

For the combinator $(a_1 \oplus a_2)$, we require that the tag of the garbage type be decided by the direction of the branch and by extension, the tag of the input. Say a_1: $b_1 \rightsquigarrow b_3$ and a_2: $b_2 \rightsquigarrow b_4$. So $(a_1 \oplus a_2)$: $(b_1 \times b_2) \rightsquigarrow (b_3 \times b_4)$. Like for **left** a, each branch must provide a value of the relevant output type i.e. $(b_3 \times b_4)$. Let the heap types of a_1 and a_2 be h_1 and h_2. Garbage types are g^1 and g^2.

Consider the left branch. For the reversible computation of a_1, it needs the default value $\phi(h_1)$. It also needs to extract the tag of its output from default value **left** $\phi(b_3)$ of type $b_3 + b_4$ using **leftSwap**. It produces as garbage a value of of type g^1 from the computation of a_1, $\phi(b_3)$ from the **leftSwap** as well as the heap values it does not need. Similarly, the right branch

needs heap values of type h_2 and $b_4 + b_3$. It will produce as garbage a value of type g^2, $\phi(b_4)$ from its **leftSwap** as well as unnecessary heap values. So we provide in the heap values needed by at least one of the two branches:

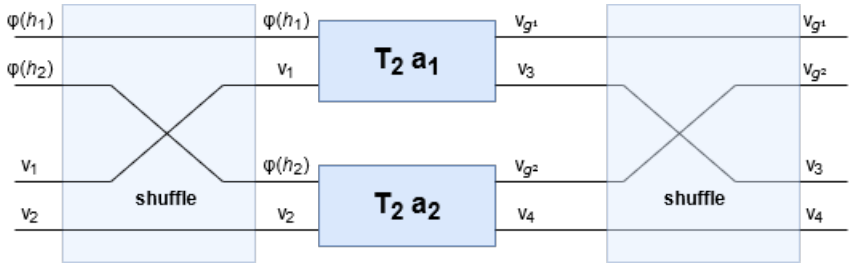
$$\text{heap}\ \{b' = b_3 + b_4\}\ (a_1 \oplus a_2) = ((\text{heap}\ a_1) \times (\text{heap}\ a_2)) \times ((b_3 + b_4) \times (b_4 + b_3))$$

And the garbage produced:

$$\begin{aligned}\text{garbage}\ \{b' = b_3 + b_4\}\ (a_1 \oplus a_2) = (((\text{garbage}\ a_1) \times (\text{heap}\ a_2)) \times (b_3 \times (b_4 + b_3))) \\ + \\ (((\text{garbage}\ a_2) \times ((\text{heap}\ a_1) \times (b_3 + b_4))) \times b_4)\end{aligned}$$

I pass in the output type as an implicit parameter (the typechecker can figure them out from the argument $(a_1 \oplus a_2)$) since the definition refers to the output types of a_1 and a_2.

The only combinator left to consider at this point is $(a_1 \otimes a_2)$. Here both a_1 and a_2 will be computed in parallel, so we need default values for both their heap types h_1 and h_2. The computations will produce garbage values of type g^1 and g^2 respectively. Implementing $\mathbf{T}_2$ $(a_1 \otimes a_2)$ is relatively straightforward:



$$\mathsf{T}_2\ (a_1 \otimes a_2) = \text{shuffle} > (((\mathsf{T}_2\ a_1) \times (\mathsf{T}_2\ a_2)) > \text{shuffle})$$

The rest of the implementation of T_2 is exactly as laid out in section 2.1.6.

3.5.1 Proof of correctness for $\mathbf{T}_2$ i.e. $(\mathbf{T}_2\ a)_\Pi\ [\phi(h),v] = [v_g,\ (a\ v)_{\mathbf{ML}_\Pi}]$

Theorem 3. *$\forall\ a : b \rightsquigarrow b'$ and $\forall v : b$, $\exists v_g : g$ such that $(\mathbf{T}_2\ a)_\Pi\ [\phi(h),v] = [v_g,\ (a\ v)_{ML_\Pi}]$ where h and g are the heap and garbage types of a.*

The theorem has been formalised as follows:

$$\begin{aligned}&\mathsf{T}_2\text{-proof} : \forall\{b_1\ b_2\}(c : (b_1 \rightsquigarrow b_2))(v : \text{val}\ b_1) \rightarrow \\ &\quad \Sigma\ (\text{val}\ (\text{garbage}(c)))\ (\lambda\ g' \rightarrow \\ &\quad ((\mathsf{T}_2\ c)\ [\ ([\ \phi(\text{heap}(c))\ ,\ v\ ])\ ]^{\mathsf{f}}) \equiv ([\ g'\ ,\ (c\ [\ v\ ]^{\mathsf{a}})\ ]))\end{aligned}$$

The statement is regarding the existence of a garbage value. Because Agda is constructive, we must provide an appropriate garbage value as a witness and a proof of the equality. The interesting part here is obtaining the witness since the proof is simply via induction on the relevant combinators. For example, for T_2 $(a_1 \ggg a_2)$, the garbage value produced is the product of the garbage values produced as a result of the computations $(a_1\ v)$ and $(a_2\ (a_1\ v))$. Similarly, from Figure 3.5, we can deduce the garbage value for T_2 **(left** a**)** when we take the **left** branch. The garbage value is everything produced in the branch other than **left** $\mathbf{v}_2$:

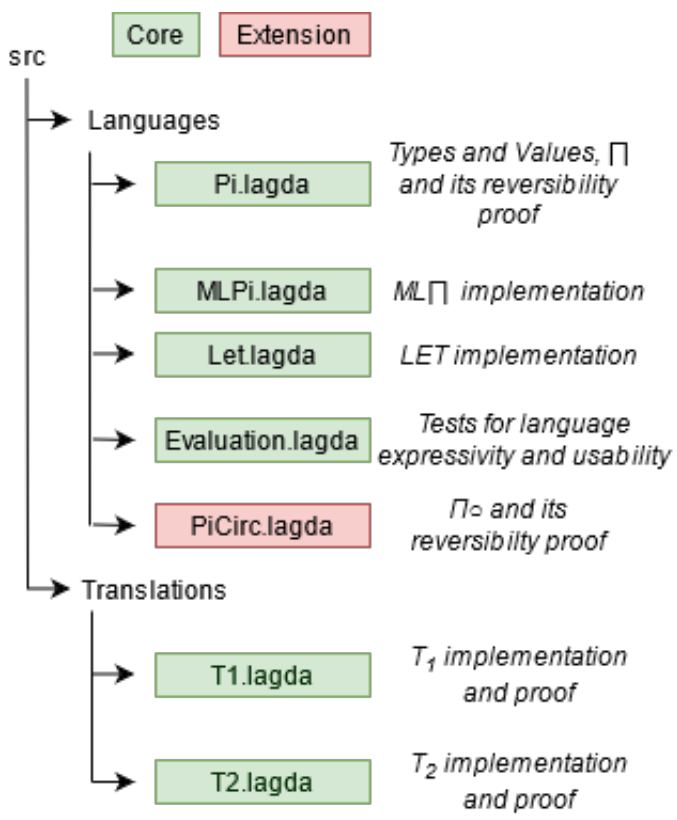
```
T2-proof (a1 >>> a2) v with (T2-proof a1 v) | (T2-proof a2 (a1 [ v ]ᵃ))
... | (g1 , pf1) | (g2 , pf2) rewrite pf1 | pf2 = ([ g1 , g2 ]) , refl
T2-proof {b2 = b' + b''} (left a) (left v) with (T2-proof a v)
... | (g , pf) rewrite pf = (left ([ g , ([ φ(b') , (left(φ(b'')))]) ])) , refl
```

3.6 Summary

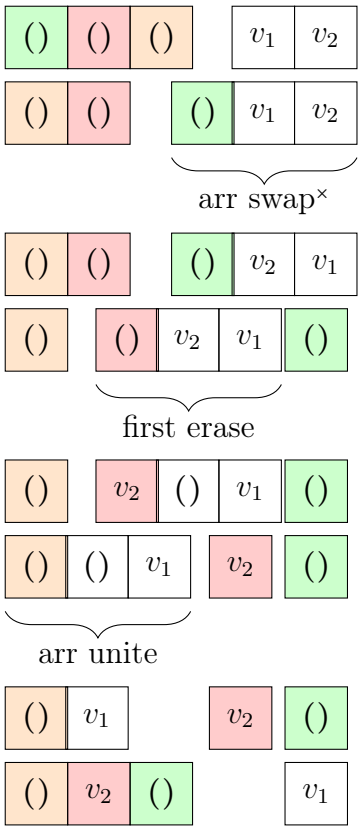
I have proved T_1 and T_2 correct: they preserve the semantics from LET to ML_Π and from ML_Π to Π respectively. Transitively combining the equivalences, the composition of T_1 and T_2 is a verified compiler from LET to Π:

Theorem 4. *($T_2 \circ T_1$) from LET to Π preserves the semantics of LET.*

3.7 Repository Overview



(a) Repository structure: Original work with 54 PiCirc.lagda lines modified from [11]



(b) T_2 on **fstA**: Heap and garbage values are coloured.

Figure 3.8

Chapter 4

Evaluation

The core deliverable in this project, a formalisation and proof of the translation ideas in James and Sabry [10] for a strongly normalising subset of the associated languages, has been completed with **no** postulates or assumptions. I also completed two extensions which extended Π to Π° and proved the latter reversible. In this chapter, I take a critical look at the usability of LET and the efficiency of the compilation. I then show that the extended language Π° is Turing-complete and lastly, I evaluate my proof techniques.

4.1 How expressive and usable is LET? How efficient is the compilation to Π?

LET is a total language, meaning it is strongly normalising and hence does not allow looping computations. It is still quite expressive, however, since it can encode arbitrary Boolean functions and by the end of this section, we will have a program that adds two n-bit numbers in LET, with the help of the metalanguage Agda in order to output a program that does this for any fixed (but arbitrary) n. We can provide the arguments in the environment since that is the input to the expression.

Consider the much simpler NOT function, whose input and output are both single bits. In the type system of this implementation, we encode booleans as the type $1 + 1$, as shown below:

```
bool = 𝟙 + 𝟙 ; true = right [] ; false = left []
```

The input bit is put in the environment, which becomes a vector of length 1, and we know that the output will be a bit. NOT is therefore expressed as follows, where $\mathbf{var_e}$ **here** refers to the 0th element in the environment:

```
NOT-test : (bool :: []) ⊢exp: bool
NOT-test = ₑcase (varₑ here) ₑL (rightₑ []ₑ) ₑR (leftₑ []ₑ)
```

The above expression encodes the truth table of NOT directly, saying that if the input has a left tag, then output true and similarly for when there is a right tag on the input. Clearly, NOT is already reversible. It does not lose any information. Indeed it is directly expressible in Π as $\mathbf{swap^+}$**: bool** $\leftrightarrow$ **bool**. So one might conjecture that there is only an empty heap generated during the translation of NOT. This is not the case because the translation simply works on the structure of a LET expression and not its meaning. We could just as easily have encoded a constant function using a similar structure.

We have arrived at a **fundamental trade-off in reversible languages**: *ad-hoc* constructs that leave it up to the programmer to ensure reversibility after taking into account the semantics versus a general approach that overestimates the information required to ensure reversibility. The former approach requires far less memory but is error-prone while the latter (as shown in this dissertation) is provably correct.

There are a few steps involved in computing the size of the heap generated during the translation of an expression. First, we need to define the size of a type. Ideally, this definition should have some link to the amount of information present in a value of that type. The empty type carries no information and neither do types composed solely of products and the empty type. The only way of transferring information is via the tag in the value of a sum type. There are two possible tag values, and hence we can say that each tag carries 1 bit of information. This gives us a relatively simple way to measure the bits of information carried in a type: it is simply the number of tags present in a value of that type. Therefore, a product type carries the sum of the bits of information from its two constituent types, a sum type carries the information from any one of its two constituent types plus one bit for the tag associated with it while the unit type carries no information at all. As calculated below:

$$\begin{aligned}
&\mathsf{size}\ \mathbb{1} = 0 \\
&\mathsf{size}\ (b_1 + b_2) = \mathsf{suc}\ (\mathsf{max}\ (\mathsf{size}\ b_1)\ (\mathsf{size}\ b_2)) \\
&\mathsf{size}\ (b_1 \times b_2) = \mathsf{size}(b_1)\ \mathsf{Data.Nat.+}\ \mathsf{size}(b_2)
\end{aligned}$$

In this evaluation, I inspect the size of the generated heap because a major problem associated with reversible execution is the memory overhead. One of the sources used for this project states that "the runtime and memory costs of reversible execution are (polynomially) larger than those of irreversible execution."[17]. For the functional language LET, which does not have side-effects, this sounds about right since we can simply return the input as well as the output after a computation. A similar upper bound is also made clear in Toffoli's theorem: the number of default bits we need will not be greater than the number of input bits. So why then, is the heap size associated with the translation of NOT, 12 bits instead of a maximum of 1 bit (the size of the single-bit input)?

The unexpected size of the heap is due to the fact that Π, the language we are compiling to, is constructed entirely out of isomorphisms. It does not admit partial bijections, which would allow a combinator **c: b ⇌ b'** to not terminate on a particular value of type b. Say we have a program P in LET, which takes an environment of type b_1 and outputs a value of b_2. Clearly, if our translation were simply to return the environment as well as the output, then we could trivially reverse the execution. However, the translation of P (call this $(T_2 \circ T_1)$ **P**) in Π must have an output value of $b_1 \times b_2$. So the reverse of $(T_2 \circ T_1)$ **P** must output a value given any input of type $b_1 \times b_2$. But what if it receives an input $[\rho, v]$ such that evaluating P on environment ρ does not output v? This computation diverges and hence is not a complete bijection. So it is not expressible in Π. This is essentially the same reason cloning is not expressible in Π. Consider a combinator that clones bits. The reverse of this combinator must diverge when given an input such as $[true, false]$.

During translation, we need to clone environments which are composed entirely of bits. Cloning sum types is non-trivial and we need to keep track of branching information during this cloning as well as in the normal course of execution. It is difficult to present an estimate of the heap size as a function of execution length, which due to the absence of loops in LET, is linear in the length of a LET expression.

The n-bit AND function was implemented in LET and next I show a graph of heap size versus n. The expression for arbitrary n was generated in Agda using the functions **and** and **and-gate**. An explanation of **replace** will follow after a discussion about the heap size complexity. It will be used to make a wider point about the usability of LET as implemented.

```
and : ∀{n : ℕ}{Γ : Vec 𝕓 n} → Γ ⊢exp: bool → Γ ⊢exp: bool → Γ ⊢exp: bool
and e₁ e₂ = ₑcase e₁ ₑL (leftₑ []ₑ) ₑR (replace zero {Γ₁ = []} e₂)

and-gate : ∀(n : ℕ) → (bool-env n) ⊢exp: bool
and-gate zero = rightₑ []ₑ
and-gate (suc n) = and (varₑ here) (replace zero {Γ₁ = []} (and-gate n))
```

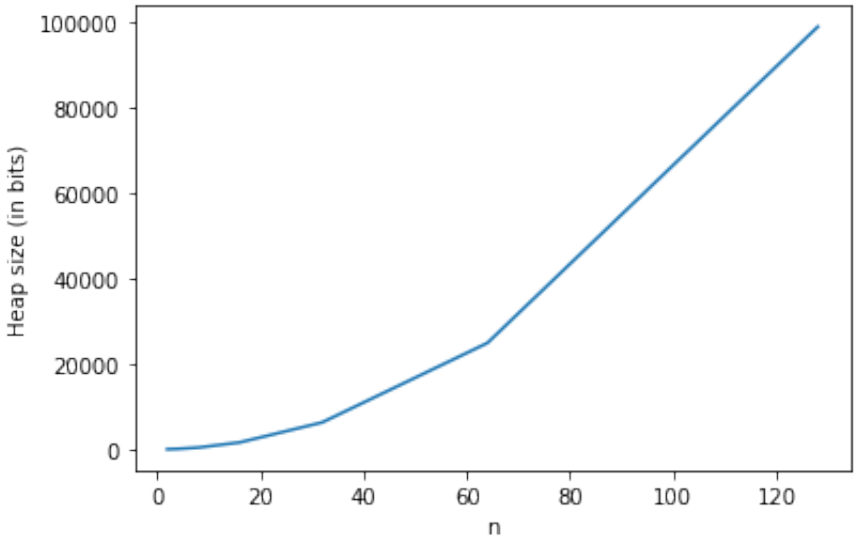


Figure 4.1: Heap size vs number of input bits n

Encouragingly, the heap size seems to grow linearly with n, albeit multiplied by a large constant. As expected, the n-bit OR function returns exactly similar results.

The only ML_Π combinators that add information to the heap are **create** and $c_1 \oplus c_2$. The amount of information added depends on the types involved and hence estimating heap size as a function of expression length becomes intractable. This will vary according to the exact LET expressions we are translating. For instance, a bigger environment will lead to larger constants in the estimate. The heap often contains useless default values in order to avoid duplication of branching information. This would not be a problem were we translating to the extended language Π°, where we can create arbitrary constants, add tag information and so on. So hopefully, the translation when extended, will obey the upper bound as given by Toffoli's theorem.

Now for a discussion about the usability of LET. Consider the **and** function above, which essentially generates the expression (e_1 **AND** e_2). A first attempt at implementing this might be **case** e_1 **L (left []) R** e_2. However, in a problem that crops up frequently when using de Bruijn indices, the references change when crossing a binding (**case** and **let** are both bindings). Now 0 will refer to the value of e_1, and any references to 0 in e_2 will refer to this instead. So what we must do is increment all the references in e_2 by 1 since the relevant bindings are now all one binding level further away. This is what **replace** does. But then why is the call **replace zero** e_2? Why must we pass in the **zero** as an argument?

Consider if e_2 was of the form **let** e **in** e'. Any reference to 0 in e' will still refer to the value of e, but the remaining bindings are now one level further away. So in e', we must change any references starting with 1. If we encounter a **case** or **let** again, any references to 0 are now referred to as 1. As before, references to 0 will point to the correct value. So we must

update any references starting with 2. In this way, we may have a need to update any reference starting with the number m for example. Every time we cross a binding, we increment every reference starting with $m + 1$ instead.

$$\begin{aligned}
&\mathsf{replace} : \forall\{n : \mathbb{N}\} \to \forall(m : \mathbb{N}) \to \{\Gamma_1 : \mathsf{Vec}\ \mathbb{b}\ m\}\{\Gamma_2 : \mathsf{Vec}\ \mathbb{b}\ n\}\{b\ b' : \mathbb{b}\} \\
&\qquad\qquad \to (\Gamma_1 +\!\!+ \Gamma_2) \vdash\!\mathsf{exp}:\ b \to ((\Gamma_1 +\!\!+ (b' :: \Gamma_2))) \vdash\!\mathsf{exp}:\ b \\
&\mathsf{replace}\ m\ (\mathsf{fst}_\mathsf{e}\ e) = \mathsf{fst}_\mathsf{e}\ (\mathsf{replace}\ m\ e) \\
&\mathsf{replace}\ m\ \{\Gamma_2 = \Gamma_2\}\ ({}_\mathsf{e}\mathsf{let}\ e_1\ {}_\mathsf{e}\mathsf{in}\ e_2) = \\
&\qquad\qquad {}_\mathsf{e}\mathsf{let}\ (\mathsf{replace}\ m\ e_1)\ {}_\mathsf{e}\mathsf{in}\ (\mathsf{replace}\ (\mathsf{suc}\ m)\ \{\Gamma_2 = \Gamma_2\}\ e_2)
\end{aligned}$$

Representative cases are shown above. The declaration of **replace** was difficult to come up with. It essentially says that if we are given an expression which would have typed in an environment of length $(m + n)$ and there is a new binding at level m, then the modified expression will type in an environment with the updated binding.

Clearly, de Bruijn indices significantly impede usability of the language but on the other hand, it is significantly easier to prove properties about numbers than about arbitrary strings. Additionally, the declarations of LET expression-generating Agda functions such as the **and-gate**, require the user to work out the type of the expression they want beforehand as a result of incorporating typing judgements into the types of LET expressions. However, additional information in types allows us to prove more properties in a formal verification, which is the main purpose of this project. There is a **trade-off to be made between usability and proof rigour**.

Just as there is a generator function that outputs an expression (e_1 **AND** e_2), we can have functions to generate **carry**(e_1, e_2, e_3) and **sum**(e_1, e_2, e_3), which are the **carry** and **sum** bits required for addition in a full-adder. We can use these as the building blocks for an n-bit full-adder. But first we must define an encoding for an n-bit number m:

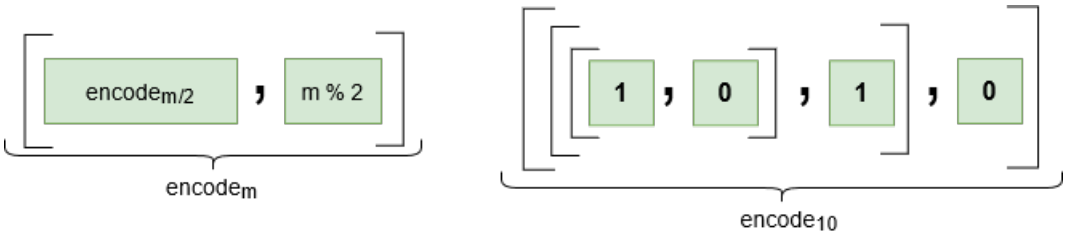


Figure 4.2: Numbers are encoded as tuples of bits. An encoding of 10 is shown.

The heap size required for the reversible execution of the full adder grows incredibly quickly with increasing n. This is likely because the environment has large types composed of sums and is therefore inefficient to clone during the many case splits required for **sum** and **carry**, which do not compute values but simply return expressions that when evaluated would return the values of the sum and carry bits. These expressions are carried forward through the generation of the expression for an n-bit full-adder and so the expression length does not grow linearly with n, so it is difficult to ascertain here whether heap size is growing linearly with expression length:

n	Heap size (in bits)
1	522
2	4647
3	28084
4	147777

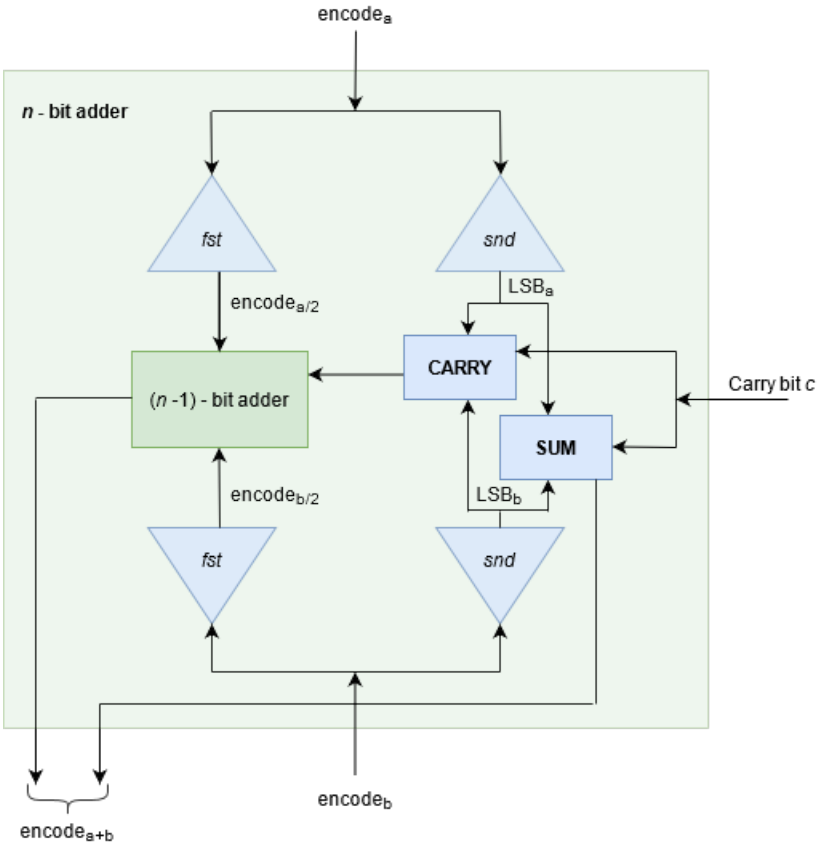


Figure 4.3: An n-bit full-adder as expressed in LET

4.2 Turing-completeness of Π°

While this section concerns Π°, I first make a claim about the expressivity of Π: we can encode **any** Boolean function within it. Appendix B constructively proves this by implementing the universal Toffoli gate in Π, thereby also proving Toffoli's theorem.

It is relatively easy to show the Turing-completeness of Π° by showing an equivalence to the Turing-complete set of partial recursive functions. This is a testament to the excellent design of Π° considering that it took about two decades to show that the *ad-hoc* reversible language Janus [15] was Turing-complete. [22] In this example inspired from [10], we can compute whether a number n is even using **trace** and the additional information provided is a default boolean value b (set equal to **true**). We then iterate **not** n times on b to obtain our final result. By recording n and this final result, we can obtain b by performing the computation backwards.

Primitive recursion can be expressed using a modification of the below combinator. Recalling that, if $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$, then the primitive recursion of f and g, which we can call $h : \mathbb{N}^{k+1} \to \mathbb{N}$, is defined as:

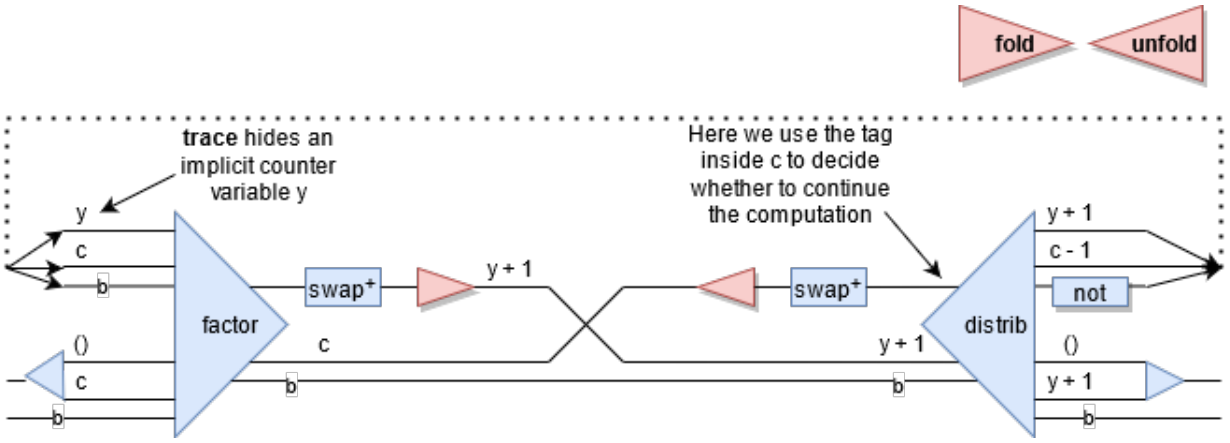


Figure 4.4: Combinator to compute the parity of a number

$h(0, \overrightarrow{x}) = f(\overrightarrow{x})$
$h(y+1, \overrightarrow{x}) = g(\overrightarrow{x}, y, h(y, \overrightarrow{x}))$

In the wiring diagram above, there is an implicit counter variable y, which after the first iteration is 0 and which we increment after every iteration. The variable c keeps track of the number of iterations left. So in the parity example, the initial value of c is n, because that is how many iterations of **not** we must do. After the first iteration, c becomes $n-1$. After the second iteration (if $n \geq 2$), it becomes $n-2$ and so on. Meanwhile, the variable b is an accumulator that holds the result of intermediate computations. This construct behaves quite like a for-loop, which is convenient because we can express primitive recursion as a for-loop too. Expanding the definition of h for a few values, we see:

$h(0, \overrightarrow{x}) = f(\overrightarrow{x})$
$h(1, \overrightarrow{x}) = g(\overrightarrow{x}, 0, h(0, \overrightarrow{x}))$
$h(2, \overrightarrow{x}) = g(\overrightarrow{x}, 1, h(1, \overrightarrow{x}))$ ...

Suppose we want to compute $h(n, \overrightarrow{x})$. The initial value passed to the combinator, instead of b, would be $\overrightarrow{x}$ and $h(0, \overrightarrow{x}) = f(\overrightarrow{x})$, since this will be an intermediate value in the computation regardless of the value of n. We must then iterate the computation n times, so $c = n$. Now we produce intermediate values $h(1, \overrightarrow{x})$, $h(2, \overrightarrow{x})$ and so on by running g. The variable y in the combinator is the counter variable required for the calculation of g when we are computing $h(y+1, \overrightarrow{x}) = g(\overrightarrow{x}, y, h(y, \overrightarrow{x}))$.

Other constructions that are required for partial recursion are: the **zero** and **successor** functions, **projection** and **composition** functions and most importantly, **minimization**. Projection functions, when expressed reversibly, simply move values around which can be achieved using **assocr**$^{\times}$, **assocl**$^{\times}$ and **swap**$^{\times}$. Composition can also be performed reversibly provided the constituent functions preserve their arguments, which is possible since we can clone numbers using **add**$_1$ and bounded iteration. The **zero** and **successor** functions are implemented using **just:** $b \rightleftharpoons (1 + b)$, which injects a value into a larger type. [10]

These constructs have all been type-checked in my implementation. To see why **just** is partial: b is a smaller type than $1 + b$. There is no input for which the output of **just** is **left** (). This partiality must be provided by **trace**, and perhaps we should use a type out of which we can potentially 'extract' a value of type 1. The type **nat** is a good candidate for this. Using **trace**, **just** takes a value v and returns **right** v. We can then use **just** to define the successor function, which we call **add**$_1$, by noting that $n + 1 =$ **right** $\langle n \rangle$. The **zero:** $1 \rightleftharpoons \mathbb{N}$ function

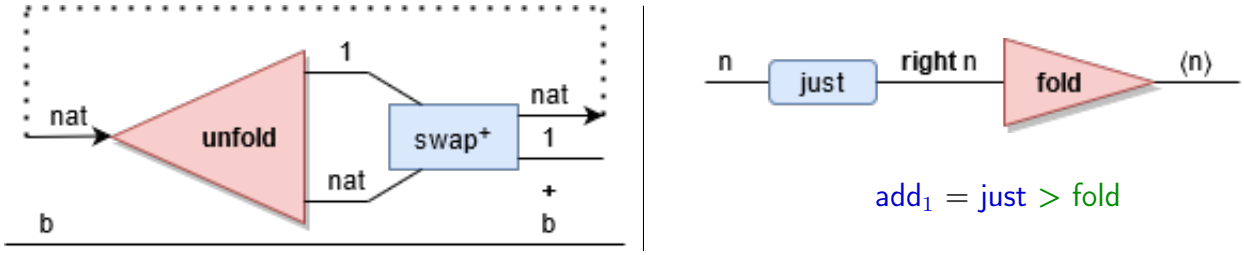


Figure 4.5: **just** on the left and $\mathbf{add}_1$ on the right

has been verified in my implementation:

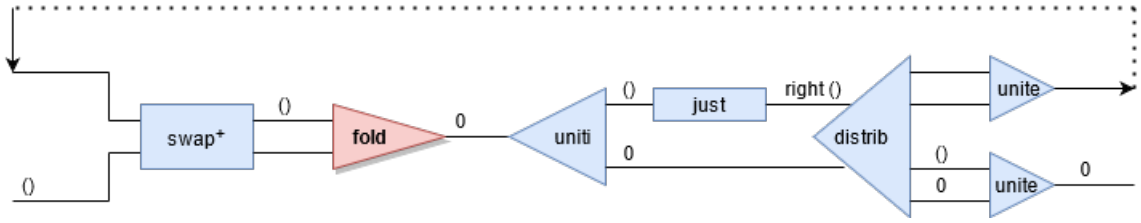


Figure 4.6: **zero** function

The last construct to implement is **minimization**. This construct has been type-checked in my implementation and is entirely original. Minimization is essentially a search through the natural numbers so just as for primitive recursion, we have a counter variable. However, instead of using the number of iterations left to decide whether to stop, we check whether we have found what we are looking for, i.e. 0. We recall that if $f : \mathbb{N}^{k+1} \to \mathbb{N}$, then the minimization of f, $\mu_f : \mathbb{N}^k \to \mathbb{N}$, is defined as:

$\mu_f(\vec{x}) = \text{least } i \geq 0 \text{ s.t } f(i, \vec{x}) = 0$

In the combinator, we remove the value generated during the computation of f using $f^\dagger$ once we have branched using it. Hence, we ensure that garbage does not build up.

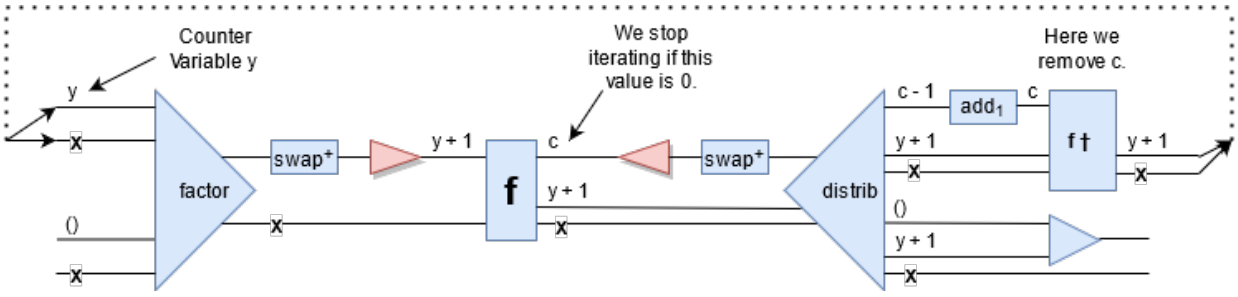


Figure 4.7: Minimization

Hence shown that the set of partial recursive functions can be expressed in Π°!

4.3 Proof definitions and techniques

Correctness of the formalisation

My core formalisation is a **complete** formalisation. I have formalised all theorems and lemmas with no postulates or axioms. The Agda typechecker has accepted these proofs. There are subtle differences, de Bruijn indices for instance, between Π as described abstractly in [10] and my formalisation. There may also be faults in the Agda interpreter itself, given that Agda

first appeared in 2007 and is still undergoing development. That being said, I am reasonably confident that my formalisation is rigorous and without error.

My proof of the reversibility of Π°, on the other hand, includes one postulate regarding the reversibility of loops. Since Agda is a total language, implementing Turing-complete Π° in Agda has involved disabling the termination checker. This still allows for some rigour in the proofs, but there may be some unwanted inconsistency introduced as a result of non-termination.

Proof techniques

I have not used equational reasoning (section 2.2) for my proofs, which would have made them more readable. Given the number of cases in these proofs, a conservative estimate is that doing so would have added at least 2000 lines to my implementation. While keeping proofs simple and readable is desirable, the focus of this project is on formalisation and verification. I have used Agda as a proof assistant, and not as a language that can 'check' pen-and-paper style proofs. The latter use is common and often prudent, but the expressions and terms that my implementation deals with are too large to avoid the problem of **congruence**. To better understand this, consider the following line from section 3.4.3:

$$\Pi\text{-rev-proof}\ (c_1 \times c_2)\ ([\ v_1\ ,\ v_2\ ]) = []\text{-cong}\ (\Pi\text{-rev-proof}\ c_1\ v_1)\ (\Pi\text{-rev-proof}\ c_2\ v_2)$$

Without additional syntax, Agda cannot arbitrarily replace instances of a with b despite being given a proof of $a \equiv b$. The implication for this case is that it cannot equate $[a_1,b_1]$ with $[a_2,b_2]$ despite being given proofs that $a_1 \equiv a_2$ and $b_1 \equiv b_2$. I defined a function []-cong that assimilates the latter two proofs to give the required equality. To gain a better understanding of the true scale of the problem, consider that some terms in my translations are of the form $a_1 \ggg a_2 \ggg \ldots \ggg a_n$ where $n \geq 15$. To expand these terms out and provide congruence functions that replace sub-terms using arbitrary equalities, for each such term, would be a Herculean yet not very formative task.

Instead I use **rewrite**, which replaces instances of the LHS with the RHS in any given goal (not necessarily an equality), when given a proof that $LHS \equiv RHS$. This has been a very formative experience since I have had to learn how exactly the typechecker deduces equality, where it falters and where I must step in and provide additional information.

In quantitative terms, the de Bruijn factor [20] (ratio of the size of a formal mathematical translation to that of an informal exposition such as a pen-and-paper proof) of my proofs will be very low. This means that proofs have been compressed to a great extent during formalisation.

Proof definitions

My use of dependent types to incorporate extra information into types has allowed implicit typing judgements using constructors and implicit type-safety proofs in function definitions.

Chapter 5

Conclusions

5.1 Contributions

Reversible computing is an important field but sadly, not well-publicised in computer science. I hope that this dissertation gives the reader a brief glimpse into reversibility and its advantages.

More importantly, this project provides a formal verification of the compilation from an irreversible to a reversible language for a strongly normalising subset of the languages. Additionally, I have programmed using category-theoretic constructs in Π° to show an equivalence between Π° and the Turing-complete set of partial recursive functions, with no additional time complexity requirements that I can immediately see. I am not aware of any similar equivalence arguments for other languages like Π°.

5.2 Extendibility

I would have liked to formally verify a translation from a Turing-complete higher-order language to Π°. However, the bottleneck proved to be not the translation itself, but the implementation of recursive types in the low-level language. At present, my implementation of Π° has recursive types able to represent natural numbers and this makes it Turing-complete.

No good general-purpose implementation of recursive types currently exists in Agda that I am aware of. Implementing them from scratch in this project, would have been another dissertation in itself and far beyond the current scope. It is the limited subtyping in the metalanguage Agda that limits the usability of current implementations. Once we have a rigorous implementation, extending LET, ML_Π, T_1 and T_2 would be relatively trivial. The proofs can be slightly modified to existence proofs to allow for partial bijections and non-termination, but I believe the core ideas will still carry through.

5.3 Lessons learnt

Through this project, I have been introduced to reversible computing and proof assistants. If I were to restart this project, I might more carefully consider whether I can fill the gap in terms of implementations of recursive types in Agda before doing the rest of my formalisation. This project has allowed me to get in touch with the more mathematical side of programming and I look forward to exploring more type theory in the future.

Bibliography

[1] Coq proof assistant. https://coq.inria.fr/.

[2] Isabelle proof assistant. https://isabelle.in.tum.de/.

[3] Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem, 1972.

[4] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015*, volume 195 of *EPTCS*, pages 71–83, 2015.

[5] Anthony Bordg, Hanna Lachnitt, and Yijun He. Certified quantum computation in isabelle/hol. *Journal of Automated Reasoning*, Dec 2020.

[6] Jacques Carette and collaborators. Collaborative work on reversible computing, 2012. https://github.com/JacquesCarette/pi-dual.

[7] 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.

[8] Michael P Frank. The r programming language and compiler. Technical report, 1997. MIT Reversible Computing Project Memo.

[9] Michael P. Frank. Reversibility for efficient computing. Technical report, 1999. PhD dissertation.

[10] Roshan P. James and Amr Sabry. Information effects. *SIGPLAN Not.*, 47(1):73–84, January 2012.

[11] Timothy Jones. Recursive types, 2016. https://github.com/zmthy/recursive-types.

[12] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014.

[13] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[14] Xavier Leroy. Formal verification of a realistic compiler. INRIA Paris-Rocquencourt.

[15] Christopher Lutz. Janus: a time-reversible language, 1986. Letter to R. Landauer.

[16] Ulf Norell. Dependently typed programming in agda. 2009. In Proceedings of the 6th International Conference on Advanced Functional Programming, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Kalyan S Perumalla. *Introduction to Reversible Computing*. CRCPress, Knoxville, Tennessee, US, 2014.

[18] 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.

[19] C. Vieri. Pendulum–a reversible computer architecture. 1995.

[20] Freek Wiedijk. The de bruijn factor. 08 2000.

[21] Roshan P James William J Bowman and Amr Sabry. Dagger traced symmetric monoidal categories and reversible programming. Technical report, 2011.

[22] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Reversible flowchart languages and the structured reversible program theorem. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 258–270, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Appendix A

Proof of Lemma 1 or the correctness of lookup

For any environment ρ of type Γ, $\rho[x] = (lookup_e\ \Gamma\ x)_{ML_\Pi}\ \rho_e{}^\times$

This lemma will be proved by induction on x, since x is a natural number less than the length of Γ. Note that here x is a de Bruijn index. Let $\rho = [0 = v, \rho']$. So $\rho_e{}^\times = [\rho'_e{}^\times, v]$.

Proof. We begin with the **base case** i.e. $x = 0$:

$$\begin{gathered}\text{lookup}_e\ \Gamma\ 0 = \textbf{sndA} \\ \textbf{sndA}\ \rho_e{}^\times = \textbf{sndA}\ [\rho'_e{}^\times, v] = v \\ \text{So } (\text{lookup}_e\ \Gamma\ 0)\ \rho_e{}^\times = v = \rho[0]\end{gathered}$$

□

Proof. Now we move on to the **inductive case** i.e. $x = n + 1$:

$$\text{lookup}_e\ \Gamma\ (n{+}1) = (\textbf{first}\ (\text{lookup}_e\ \Gamma\ n)) \ggg \textbf{fstA} \tag{a}$$

From (a) and the semantics of **first**, $\ggg$ and **fstA**, it follows that:

$$(\text{lookup}_e\ \Gamma\ (n{+}1))\ \rho_e{}^\times = \textbf{fstA}\ ((\textbf{first}\ (\text{lookup}_e\ \Gamma\ n))\ [\rho'_e{}^\times, v]) = (\text{lookup}_e\ \Gamma\ n)\ \rho'_e{}^\times \tag{b}$$

By induction, it follows that:

$$(\text{lookup}_e\ \Gamma\ n)\ \rho'_e{}^\times = \rho'[\text{n}] \tag{c}$$

$\rho[n + 1] = \rho'[n]$. Using this fact, as well as (b) and (c), we have:

$$(\text{lookup}_e\ \Gamma\ (n{+}1))\ \rho_e{}^\times = (\text{lookup}_e\ \Gamma\ n)\ \rho'_e{}^\times = \rho'[\text{n}] = \rho[n + 1]$$

□

This lemma has been formalised in Agda as follows:

```
var-proof : ∀{n : ℕ}{Γ : Vec 𝕓 n}{x : Fin n}{b : 𝕓} → (ρ : Γ env) → (m : (Γ [ x ]= b))
                    → ((_[_]) ρ m) ≡ ((lookupₑ Γ m) [ ((ρ)ₑ^×) ]ᵃ)
var-proof (ρ +ₑ v) here = refl
var-proof (ρ +ₑ v) (there m) = var-proof ρ m
```

Appendix B

Construction of the Toffoli-gate in Π

This construction has been taken from [10].

```
- Function to generate if c for any appropriate combinator c
if : ∀{b} → b ↔ b → (bool × b) ↔ (bool × b)
if c = distrib > ((id + (id × c)) > factor)

- Defining cnot
cnot : (bool × bool) ↔ (bool × bool)
cnot = if NOT

- Defining the Toffoli gate
Toffoli-gate : (bool × (bool × bool)) ↔ (bool × (bool × bool))
Toffoli-gate = if cnot

- Proof that gate works as expected
Toffoli-proof : ∀{v₁ v₂ : val bool} →
                Toffoli-gate [ [ v₁ , [ v₂ , (right []) ] ] ]ᶠ ≡ [ v₁ , [ v₂ , (NAND v₁ v₂) ] ]
```

Project Proposal

2433E

Computer Science Tripos Part II Project Proposal

Verified Compilation to A Reversible Language

May 14, 2021

Project Originator: Prof. Timothy G. Griffin

Project Supervisor: Prof. Timothy G. Griffin

Signature:

Director of Studies: Dr John K. Fawcett

Signature:

Overseers: Dr Sean B Holden and Dr Neel Krishnaswami

Signatures:

Introduction and Motivation

Compilers are part of the critical infrastructure in today's digital world. And sometimes, a compiler can introduce bugs in translation. Projects like CompCert and CakeML attempt to solve this problem by building verified compilers for C and Standard ML respectively.

This project will build a verified compiler for translation from a small irreversible language to a small reversible language. Both languages are defined in 'Information Effects'.[10].

Reversibility is inspired by low-energy computing. A reversible target language would be potentially useful for writing embedded software for energy-efficient devices.[9]

Description of the Work

The project will be written in the proof assistant Agda. It is a dependently typed language, which means that types can be indexed by arbitrary values not known at compile time. Types can also be passed to functions as parameters and returned as return values.

There are three major challenges in this project:

1. To get familiar with Agda
2. To express all proofs in a constructive way since Agda is based on intuitionistic type theory
3. Since the paper is presented informally at a high-level, it neglects to present the details of some definitions and proofs - these will have to be worked out in my project.

Since types can depend on values in Agda, we can encode propositions as types, whose elements are proofs of the corresponding propositions. So we can write code for the associated languages or the compiler and a correctness proof at the same time. The following code is an example implementation of **swap** and a proof of its reversibility in Agda.

The data type shown below is indexed by the values of combinators, labelled **comb** and the values of the language Π. It is used here to define an operational semantics for Π. **swap** is a common primitive found in reversible programs.

```
data _ _ ↦ _ : comb → Π → Π → Set where
  swap: ∀{v₁ v₂ : Π}
        -----------
        → swap (v₁,v₂) ↦ (v₂,v₁)

-- To prove: swap v ↦ v' → swap v' ↦ v

swap-proof: ∀{v v' : Π} → swap v ↦ v' → swap v' ↦ v
swap-proof swap = swap
```

Function types are used for implication. f:A → B will return proof of B given proof of A. **swap-proof**, given proof of swap v ↦ v', will return proof that swap v' ↦ v. Both these types have only one possible constructor **swap**. On the LHS, when Agda sees swap, It unifies (v_1, v_2) with v and (v_2, v_1) with v'. So it is expecting a return value of type swap $(v_2, v_1) \mapsto (v_1, v_2)$ and swap is valid constructor of this type. So this function type-checks.

This project will closely follow [10]. The paper defines a model of reversible computation that we can translate irreversible languages to. A reversible language is one where every program construct c has an inverse $c^\dagger$ such that $c\ v \mapsto v' \iff c^\dagger\ v' \mapsto v$. This means we

cannot create or destroy information. Unlike other reversible languages which are imperative, the language in [10] is functional and builds upon type isomorphisms.

The paper starts by defining a strictly normalising low-level language called Π. There will be an implementation of the syntax and semantics of this small language along with a proof of its type safety and reversibility. The paper then defines a small first-order language called LET that is strictly normalising. The collected definitions of Π and LET are shown below. LET will also be implemented in this project. The core deliverable will be a compiler from LET to Π together with a proof of its correctness.

The paper only defines a small-step semantics for the languages and does not give any algorithms to obtain the inverse of a program construct. As a part of the above core deliverable, both big-step and small-step semantics will be defined and there will be a proof of their semantic equivalence. The implementation of Π will also contain a function to compute inverses.

Both figures shown below are from [10].

DEFINITION 3.1. *(Syntax of* Π*) We collect our types, values, and combinators, to get the full language definition.*

$$
\begin{array}{rcl}
\textit{value types}, b & ::= & 1 \mid b + b \mid b \times b \\
\textit{values}, v & ::= & () \mid \textit{left}\ v \mid \textit{right}\ v \mid (v, v) \\
 & & \\
\textit{comb. types}, t & ::= & b \leftrightarrow b \\
\textit{iso} & ::= & \textit{swap}^{+} \mid \textit{assocl}^{+} \mid \textit{assocr}^{+} \\
 & \mid & \textit{unite} \mid \textit{uniti} \\
 & \mid & \textit{swap}^{\times} \mid \textit{assocl}^{\times} \mid \textit{assocr}^{\times} \\
 & \mid & \textit{distrib} \mid \textit{factor} \\
\textit{comb.}, c & ::= & \textit{iso} \mid \textit{id} \mid \textit{sym}\ c \mid c \fatsemi c \mid c + c \mid c \times c
\end{array}
$$

By design, every program construct $c : b_1 \leftrightarrow b_2$ has an adjoint $c^{\dagger} : b_2 \leftrightarrow b_1$ that works in the other direction. Given a program $c : b_1 \leftrightarrow b_2$ in Π, we can run it by supplying it with a value $v_1 : b_1$. The evaluation rules $c\ v_1 \mapsto v_2$ for the primitive isomorphisms are given below:

$$
\begin{array}{rlcl}
\textit{swap}^{+} & (\textit{left}\ v) & \mapsto & \textit{right}\ v \\
\textit{swap}^{+} & (\textit{right}\ v) & \mapsto & \textit{left}\ v \\
\textit{assocl}^{+} & (\textit{left}\ v_1) & \mapsto & \textit{left}\ (\textit{left}\ v_1) \\
\textit{assocl}^{+} & (\textit{right}\ (\textit{left}\ v_2)) & \mapsto & \textit{left}\ (\textit{right}\ v_2) \\
\textit{assocl}^{+} & (\textit{right}\ (\textit{right}\ v_3)) & \mapsto & \textit{right}\ v_3 \\
\textit{assocr}^{+} & (\textit{left}\ (\textit{left}\ v_1)) & \mapsto & \textit{left}\ v_1 \\
\textit{assocr}^{+} & (\textit{left}\ (\textit{right}\ v_2)) & \mapsto & \textit{right}\ (\textit{left}\ v_2) \\
\textit{assocr}^{+} & (\textit{right}\ v_3) & \mapsto & \textit{right}\ (\textit{right}\ v_3) \\
\textit{unite} & ((), v) & \mapsto & v \\
\textit{uniti} & v & \mapsto & ((), v) \\
\textit{swap}^{\times} & (v_1, v_2) & \mapsto & (v_2, v_1) \\
\textit{assocl}^{\times} & (v_1, (v_2, v_3)) & \mapsto & ((v_1, v_2), v_3) \\
\textit{assocr}^{\times} & ((v_1, v_2), v_3) & \mapsto & (v_1, (v_2, v_3)) \\
\textit{distrib} & (\textit{left}\ v_1, v_3) & \mapsto & \textit{left}\ (v_1, v_3) \\
\textit{distrib} & (\textit{right}\ v_2, v_3) & \mapsto & \textit{right}\ (v_2, v_3) \\
\textit{factor} & (\textit{left}\ (v_1, v_3)) & \mapsto & (\textit{left}\ v_1, v_3) \\
\textit{factor} & (\textit{right}\ (v_2, v_3)) & \mapsto & (\textit{right}\ v_2, v_3)
\end{array}
$$

The semantics of composition combinators is:

$$
\frac{}{\textit{id}\ v \mapsto v} \qquad \frac{c^{\dagger}\ v_1 \mapsto v_2}{(\textit{sym}\ c)\ v_1 \mapsto v_2} \qquad \frac{c_1\ v_1 \mapsto v \quad c_2\ v \mapsto v_2}{(c_1 \fatsemi c_2)\ v_1 \mapsto v_2}
$$

$$
\frac{c_1\ v_1 \mapsto v_2}{(c_1 + c_2)\ (\textit{left}\ v_1) \mapsto \textit{left}\ v_2} \qquad \frac{c_2\ v_1 \mapsto v_2}{(c_1 + c_2)\ (\textit{right}\ v_1) \mapsto \textit{right}\ v_2}
$$

$$
\frac{c_1\ v_1 \mapsto v_3 \quad c_2\ v_2 \mapsto v_4}{(c_1 \times c_2)\ (v_1, v_2) \mapsto (v_3, v_4)}
$$

The syntax of LET is given below:

$$
\begin{array}{rcl}
\textit{Base types}, b & = & 1 \mid b + b \mid b \times b \\
\textit{Values}, v & = & () \mid \textit{left}\ v \mid \textit{right}\ v \mid (v, v) \\
\textit{Expressions}, e & = & () \mid x \mid \textit{let}\ x = e_1\ \textit{in}\ e_2 \\
& \mid & \textit{left}\ e \mid \textit{right}\ e \\
& \mid & \textit{case}\ e\ x.e_1\ x.e_2 \\
& \mid & \textit{fst}\ e \mid \textit{snd}\ e \mid (e, e) \\
\textit{Type environments}, \Gamma & = & \epsilon \mid \Gamma, x : b \\
\textit{Environments}, \rho & = & \epsilon \mid \rho; x = v
\end{array}
$$

The translation for LET to Π is done via an arrow metalanguage ML_Π that makes information creation and destruction evident via operations **create** and **erase**. Implementation will be done alongside proof of properties such as the correctness of the translation of a LET type environment.

Extension work: Π is a subset of Π°, which allows partiality by including looping computations that may not terminate. Implementing the additional features of Π° will be an extension. LET is a subset of a language LET°, which is Turing-complete and admits partial functions by including **for** loops. Similar to Π and Π° above, implementing syntax, semantics and typing rules for LET° will be an extension. The compiler that translates LET° to Π° will be another possible extension. Any extension work will include proofs of its correctness.

Evaluation: This project is quite proof-heavy and when it is just getting started, various proof techniques will have to be explored. For instance, a significant decision to make here is how much the proofs will rely on dependent types. At the beginning of the project, evaluation will involve some comparison between different techniques considered, perhaps by taking a small lemma and proving it using each of them. Towards the end of the project, I will consider how the proofs of some major propositions might have been different had I chosen some other proof strategies versus the one used in the project.

As an extension, the code for the languages and compiler will be compiled down to Haskell (Agda is implemented in Haskell) and a small parser will be written in Agda. A collection of programs and algorithms will be written in the source and target languages. The major goal during this process will be to test that programs in the target language are actually reversible by running the Haskell executables.

Starting Point

- **Agda:** This is a new language to me that I will learn during the course of this project. I have set time aside for this. Over the summer, I have completed a few exercises from Part 1 of *Programming Language Foundations in Agda* by Philip Wadler.
- **Reversible Languages:** I have no experience with reversible languages, and my only knowledge of them comes from reading Chapters 1, 6 and 8 of the book *Introduction to Reversible Computing*. Extensive knowledge of this area is not required for this project as the paper 'Information Effects' clearly defines any languages implemented in this project.
- **Verified Compilers:** I have no experience writing compilers and all my knowledge of them comes from the IB Compiler Construction course. I have no prior knowledge of verified compilers.

Success Criteria

This project will be deemed a success if the following criteria are met:

1. A high-level source language similar to LET and a low-level target language similar to Π have been implemented and the target language has been shown to be reversible.

2. An intermediate metalanguage similar to ML_{Π} has been implemented and its combinators have been proved correct.

3. The compiler correctly translates programs from source to target. The proof of correctness for the compiler should compile, which is indication that it is a valid proof.

The following features can be added to give Turing-complete language definitions similar to LET° and Π°. These are potential extensions and can be evaluated in the same way as core deliverables:

1. **Looping computations:** One of the additional features present in LET° and Π° is support for loops. This allows the languages to admit partial functions.

2. **Natural numbers:** LET° allows basic operations on natural numbers, and Π° has isorecursive types to support this.

3. Associated compilation of the above features

Timetable and Milestones

The dissertation will be written in parallel with the implementation and evaluation.

1. *3rd October – 16th October*

   Deadlines

   - Phase 1 Proposal Deadline – 12th October, 3 PM.
   - Draft Proposal deadline – 16th October, 12 noon.

2. *17th October – 30th October*

   Finish exercises from first 5 chapters of *Programming Language Foundations in Agda* Part 2. Read [10] again and explore possible ways to implement Π and LET. Learn to use the MCS and install required software on it (for back-ups).

   Prep for CatTheory unit graded exercise sheet - category theory is actually relevant to this project because it motivated the development of the reversible language Π

   Milestone: Send supervisor a diagram outlining implementation ideas.

   Deadlines

   - Proposal Deadline – 23rd October, 12 noon.

3. *31st October – 13th November*

   Complete CatTheory unit graded exercise sheet - I am setting aside 4 days of revision + work, though it is unlikely that it will take this long

   Implement Π and prove that it is reversible and type-safe. Based on the lessons learned and difficulties encountered, draw up an implementation plan for the project. Explore proof strategies and ways to obtain low-level compiled code.

Milestone: The proof of reversibility compiles and the implementation plan has been sent to my supervisor.

Dissertation: Preparation Chapter: Background and Motivation + include relevant bits about Agda/dependent type theory. Outline how to compile down to Haskell and reasons for doing so in evaluation chapter.

Other deadlines

- Category Theory Ex Sheet 4 – 6th November, 4pm.

4. *14th November – 27th November*

   Implement language ML_{Π} in Agda. If time permits, add additional features in Π for reversible arithmetic, inspired by other reversible languages.

   Dissertation: Preparation Chapter: Write down major steps taken for implementation to start. Implementation Chapter: Explain design decisions for implementation of Π and ML_{Π}, Evaluation Chapter: Log steps to be taken for demonstration for Π's reversibility.

5. *28th November – 11th December*

   Implement combinators for ML_{Π} such as for cloning, erasing and joining. Prove their correctness. Outline 3 benchmark programs to test out in Π.

   Milestone: Confirm that ML_{Π} follows the definition in [10].

   Dissertation: Implementation chapter: Major design decisions made in this stage. Evaluation chapter: Write about benchmark programs to be used to test out Π and outline why they were chosen.

   Other deadlines

   - Category Theory Take-home test – 30th November, 4pm - 4th November, 4pm

6. *12th December – 25th December*

   Implement LET abstract syntax, semantics and typing rules. (Implementation outline would be completed in a previous work package).

   Milestone: All language implementations for core deliverables complete.

   Dissertation: Log relevant implementation details and strategy for testing out LET in evaluation chapter.

7. *26th December – 8th January*

   Exam revision and contingency

   If desired, extend implementations above to LET° and Π° (and prove reversibility). In the evaluation chapter, outline how they are to be tested, similar to that for core deliverables.

8. *9th January – 22nd January*

   Write a simple recursive descent parser in Agda to start testing the low-level compiled code for Π, ML_{Π} and LET. Complete any tests previously planned.

Translate LET to intermediate language ML_{Π} and prove that the semantics have been preserved. This corresponds to lemmas 8.1 and 8.2 from [10]. Obtain low-level compiled code for this translation.

Milestone: Finish proof of correctness. Experimental demonstration of translation using Agda's evaluation mechanisms.

Dissertation: As before, log implementation details and strategy for evaluation of various components. Log any relevant results obtained from testing. Read through Preparation chapter and add any relevant preparatory steps that were unforeseen. Complete first draft of Preparation chapter and send to supervisor.

9. *23rd January – 5th February*

   Prepare progress report and send to supervisor. Translate intermediate language ML_{Π} to Π and prove that the semantics have been preserved. This corresponds to lemma 9.1 from [10]. As in previous slot, obtain low-level compiled code.

   Milestone: Proof of correctness compiles. Experimental demonstration of translation.

   Dissertation: Implementation of core deliverables will be complete at this stage. Log relevant details.

   Deadlines

   - Progress Report Deadline – 5th February, 12 noon.

10. *6th February – 19th February*

    Prepare progress report presentation. Evaluate translations using 3 benchmark programs. If time permits, extend implementation to ML_{Π° and translate LET$^\circ$ to it.

    Milestone: Evaluation of core deliverables complete.

    Dissertation: Complete implementation and evaluation chapters since the core of this project is now complete.

    Deadlines

    - Progress Report Presentations – 11th, 12th, 15th, 16th February, 2 PM.

11. *20th February – 5th March*

    Slot reserved for potential unit of assessment work

    If time permits, translate from ML_{Π° to Π° and prove correctness.

12. *6th March – 19th March*

    Slot reserved for potential unit of assessment work

    If time permits, continue working on extensions and compile code down to Haskell. Evaluate using more interesting programs such as a Fibonacci generator.

    Milestone: Evaluation of extensions complete.

    Dissertation: Add details of extensions made to implementation and evaluation chapters.

13. *20th March – 2nd April*

    Slot reserved for mostly exam revision

    Write introduction and conclusion chapters and send first draft of dissertation to supervisor.

14. *3rd April – 16th April*

    Slot reserved for mostly exam revision

    Make any required changes to dissertation and send to DoS and supervisor.

15. *17th April – 30th April*

    Slot reserved for mostly exam revision

    Check that all details are filled in properly, that the dissertation is formatted properly, run through spell-checker. Clean up source code.

16. *1st May – 14th May*

    Slot reserved for mostly exam revision

    Submit dissertation and source code.

    Deadlines

    - Dissertation Deadline – 14th May, 12 noon.
    - Source Code Deadline – 14th May, 5 PM.

Resources Declaration

I will be using my personal laptop: A Lenovo IdeaPad S340. Specifications: Intel i5-1035G1(x86, 1GHz, 4 cores), 8GB RAM, Windows 10.

This will be used for writing and compiling the project code, as well as for evaluation. I will push my code to GitHub once a day and will store my dissertation in Google Drive. My files are automatically synced with OneDrive.

In the event of hardware failure, I have another personal laptop that I can use. I have also set aside time to learn how to use the MCS, and will ensure that the required software (emacs, stack, git...) is installed on there.