THE COMPUTATIONAL CONTENT OF ISOMORPHISMS

Roshan P. James

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of School of Informatics and Computing
Indiana University
April 2013

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

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

Amr Sabry, Ph.D.

Amal Ahmed, Ph.D.

R. Kent Dybvig, Ph.D.

Larry Moss, Ph.D.

June 6, 2012

Knox: "Forgive me for asking a crass and naive question —
but what is the point of devising a machine that cannot be built in
order to prove that there are certain mathematical statements
that cannot be proved? Is there any practical value in all this?"
Turing: "The possibilities are boundless."
– Breaking the Code

Preface

Is there a unified theory of computational effects? This question was the object of my curiosity for much of my time as a graduate student. The word *effect* is semi-formally used in much of programming language theory even though it has no definition. Does such a definition exist?

The folklore on computational effects in the functional languages community describe effects as things that breaks referential transparency (or some similar substitutivity argument) or that they break equational reasoning of the λ-calculus or that they violate the functional property (in that λ-calculus functions do not behave like set-theoretic functions anymore) and variations of these themes.

However, none of these *ad hoc* arguments hold under scrutiny. For the first, see Søndergaard and Sestoft [1990], who formalize referential transparency. They start from the original remarks made by Quine [1953] in the context of natural language and describe its import into computer science by Strachey [2000] and Stoy [1977]. Søndergaard and Sestoft [1990] compare various notions of referential transparency, unfoldability and related notions, showing why these are largely orthogonal to effects.

Second, defining effects in terms of not respecting equational reasoning does not work very well in two ways. If we add operational rules for, say, assignments to the λ-calculus, it is by definition not the λ-calculus anymore. In fact, if the resulting system is inconsistent it is also not a calculus anymore and hence it is unclear what has been achieved. However, if we were to correctly add an equational notion of assignments, we do get an extended calculus, i.e. λ-calculus with assignments and λ-calculus with continuations have been developed [Felleisen et al., 1987, Felleisen and Hieb, 1992a] where equational reasoning holds.

Third, there is nothing in the syntactic theory of the λ-calculus (neither in the call-by-name nor the call-by-value variants [Plotkin, 1975]) that require them to behave like set-theoretic functions and there is no *a priori* reason to expect set-theoretic notions to hold in the system of equalities that is the λ-calculus. Stated differently, expecting a set-theoretic notion of functions to be respected by certain λ-calculus terms is the same as expecting as expecting any other arbitrarily chosen property to hold in the λ-calculus. There is no reason why it should hold.

That said, all sorts of computational phenomena are called effects: assignments, jumps, continuations, mutable state of various sorts, recursion, non-termination, non-determinism and many such things came under the umbrella of effects including some fairly exotic devices such as 'parallel or' operator of Plotkin [1977]. Various other things behaved like effects even though they are not usually classified as effects such as the time takes for program execution, heat-dissipated by the execution of a program and such. Even purely semantic concerns seemed "effectful" in nature such as the evaluation of so called higher order contracts and redefinition of symbols in a REPL.

Commonality between these varied things, if any, is not obvious. In some cases effects were things that had access to some hidden information. For example, mutable state has the store as the hidden information, non-determinism has access to a secret coin-toss, parallel-or has access to an oracle that predicts the termination of its arguments, heat dissipation and time have access to some heat signature data or to the clock and so on. However at any point in program execution a vast number of changes occur and a theory that identifies any number of these as effects seems useless. Sabry [2008] explains this in terms of the situation of consuming a pain-killer that also thins the blood. If the intention of taking the pain-killer was indeed to thin the blood, then it is not considered a side effect. Can we build a theory of effects that classifies what things are observable in a computation and what experiments an observer (say the program itself) is allowed to make? This may partly address the issue of "hidden information" effects. If the hidden information is observable then anything that changes it should be considered effectful. That said, it is unclear how such a formal syntactic theory of observers and observables should be built. More importantly, this line of reasoning did not readily apply to many effects such as grabbing/invoking continuations and jumps where there was no simple notion of hidden information.

The term effect is commonly used in the sense of an action rather than a pervasive property. Consequently, for me, a theory of effects needed to have an operational flavor. Any explanation that was purely denotational would be unsatisfying. I was less concerned with abstracting effects and more concerned about explaining them. In the words of *The Meditations*, “This thing, what is it in itself, in its own constitution? What is its substance and material? And what is its causal nature? And what is it doing in the world? And how long does it subsist?” Was there a unified way to look at all effects? And if such a theory existed, what questions could it answer?

Compelling work had been done that hinted that a unified notion of effects existed. Moggi’s work on monads as an approach to structuring denotational semantics [Moggi, 1989, 1991] led to a pragmatic means of encapsulating effects in programming practice [Wadler, 1992]. Filinski [1996] showed that monadic effects can be captured in direct style using operators derived from delimited continuations. Several other intermediate structures had been discovered for encapsulating effects that did not naturally fit into the monadic framework such as arrows [Hughes, 2000], idioms and applicative functors [Mcbride and Paterson, 2007]. At the time when I obsessed over these matters I was largely unaware of the work on the algebraic theory of effects by Plotkin, Power, Pretnar and others [Bauer and Pretnar, 2012, Plotkin and Power, 2002, 2004, Plotkin and Pretnar, 2009].

Probably the paper that influenced my ideas the most was a lesser known document, Sabry [1998]. Here Amr Sabry tries to propose a definition for what a ‘purely functional language’ is, i.e. what it means to not have any effects. Through a series of thought experiments, the paper proposes that a pure language must be invariant under call-by-value, call-by-name and call-by-need evaluation strategies. The paper was essentially an analysis of purity and effects from Plotkin-Felleisen point of view of seeing computation as syntactic theories and their equivalences [Felleisen and Hieb, 1992b, Plotkin, 1975]. In other words, Sabry chose to propose one notion of equivalence as canonical (somewhat generously choosing a join of the most popular three calculi for λ-terms) and defining effects as everything that fell outside this space.

While there is no compelling reason to chose the equalities of Sabry [1998] as the "right" ones, the underlying idea seemed sound – i.e. the paper was proposing to define purity by introducing a canonical notion of equivalence. Or in other words, *effects arise from ad hoc choices of equalities*. Instead of accepting β, β_v or η relations *a priori* as the basis of our computational equivalences, can we have a more principled approach to the equalities that we chose in our computational model? But what do we mean when we talk of a principled approach to equalities? The most primitive notion of equality that we have is the notion of isomorphism. Even before we define a theory that can be parametrized observability, it seemed imperative to study what computational content remains if we only admit only primitive equalities like isomorphisms of simple values. Thus attention was drawn to the computational content of the isomorphisms of finite types. It came as a surprise that even basic isomorphisms like identity, commutativity, associativity and distributivity already contained much of the structure required to express computation and the resulting computational model had several interesting properties. What we found was a startlingly beautiful world of computation, computation in a very pure form, with connections to a several areas in semantics, logic, type theory, quantum physics, information theory, thermodynamics and more. This was the origin of the ideas that lead to this thesis.

Let me end with a few words of thanks. This thesis and the ideas in it would have been impossible without my advisor Amr Sabry. I owe him not just for the ideas and the continuous interaction that made this development possible, but I also owe him for teaching me how to think about computation and its nature. This is not a debt than can be easily stated in words. The other person without whom this thesis would have never happened is my wife Pooja Malpani who through the many years and the long distance between Bloomington, Indiana and Seattle, Washington separating us, was a constant source of inspiration and support. I have to mention Kent Dybvig, Larry Moss and Amal Ahmed for their wonderful classes and interaction that helped a lot of ideas develop, to Dan Friedman for his advanced problem solving class (B621) and to Esfandiar Haghverdi who introduced me to category theory.

This thesis would not have been possible without an interesting peer group and close friends with whom ideas could discussed. Especially Kyle Ross and the brunch conversations about truth, logic and computation at Le Petit Cafe every Sunday for several years and Michael D. Adams who I shared several classes and office space with. I must mention several folk at the Indiana University Programming Languages group (pl-wonks) who provided wonderful camaraderie – Ron Garcia, Abdulaziz Ghuloum, Ramyaa Ramyaa, Abhijit Mahabal, Andy Keep, Michel Salim and many others. Thank you.

Roshan P. James

THE COMPUTATIONAL CONTENT OF ISOMORPHISMS

Abstract models of computation, such as Turing machines, λ-calculus and logic gates, allow us to express computation without being concerned about the underlying technology that realizes them in the physical world. These models embrace a classical worldview wherein computation is essentially irreversible. From the perspective of quantum physics however, the physical world is one where every fundamental interaction is essentially reversible and various quantities such as energy, mass, angular momentum are conserved. Thus the irreversible abstractions we choose as the basis of our most primitive models of computing are at odds with the underlying reversible physical reality and hence our thesis:

> By embracing irreversible physical primitives, models of computation have also implicitly included a class of computational effects which we call information effects.

To make this precise, we develop an information preserving model of computation (in the sense of Shannon entropy) wherein the process of computing does not gain or lose information. We then express information effects in this model using an arrow meta-language, in much the same way that we model computational effects in the λ-calculus using a monadic metalanguage. A consequence of this careful treatment of information, is that we effectively capture the gap between reversible computation and irreversible computation using a type-and-effect system.

The treatment of information effects has a parallel with open and closed systems in physics. Closed physical systems conserve mass and energy and are the basic unit of study in physics. Open systems interact with their environment, possibly exchanging matter or energy. These interactions may be thought of as effects that modify the conservation properties of the system. Computations with information effects are much like open systems and they can be converted into pure computations by making explicit the surrounding information environment that they interact with.

Finally, we show how conventional irreversible computation such as the λ-calculus can be embedded into this model, such that the embedding makes the implicit information effects of the λ-calculus explicit.

Amr Sabry, Ph.D.

Amal Ahmed, Ph.D.

R. Kent Dybvig, Ph.D.

Larry Moss, Ph.D.

Contents

1 Introduction

“Turing hoped that his abstracted-paper-tape model was so simple, so transparent and well defined, that it would not depend on any assumptions about physics that could conceivably be falsified, and therefore that it could become the basis of an abstract theory of computation that was independent of the underlying physics. ‘He thought,’ as Feynman once put it, ‘that he understood paper.’ But he was mistaken.

– The Fabric of Reality

Deutsch [1997]

1. Equalities

The essence of our thesis is this: *to speak of equality is to speak of computation*. Notions of equality are notions of computation. This view is partly familiar from the study of various systems of equalities, called *calculi*, as models of computation. The λ-calculus, for instance, is a system of equalities over a set of syntactic λ-terms such that they result in a model of computation. We distill the idea that equality entails computation to its most primitive form. We show that there is computational content not just in specific calculi, such as the λ-calculus, but even in our most primitive notion of equality – isomorphisms of finite sets.

The equalities we speak of are not *a priori* extensional notions, but are intensional specifications. To talk about two things being equal, one has to show *how they are equal* wherein this *how* is an intensional definition of a translation from witnesses of the first to witnesses of the second and vice versa. In other words, they are programs. For us, equivalences that do not have operational content do not exist.

One may ask, what is the point of such a rarefied spartan view of computation? How does this compare with more conventional models of computation? *What is the point?*

(1) The result is computation in its purest form: Programs written in this model are witnesses of equalities. In the process of computing these programs neither gain nor lose information. We get a crisp notion of quantitative information content of programs and show that *computation preserves information*.

(2) Computation closely reflects the physical reality suggested by quantum physics: From the perspective of quantum physics, the physical world is one where every fundamental interaction is essentially reversible and various quantities such as energy, mass, angular momentum are conserved. Computation is no longer at odds with physics. Conservation of information and logical reversibility are intrinsic properties of computation as well. Further, various category theoretic models such as dagger symmetric traced monoidal categories studied for quantum physics are also the categorical models for computation based on isomorphisms.

(3) Irreversibility is a computational effect: By embracing irreversible physical primitives, models of computation have also implicitly included a class of computational effects which we call *information effects*. A consequence of our careful treatment of information, is that we effectively capture the gap between reversible computation and irreversible computation using a type-and-effect system.

Even though they seem unrelated, these connections stem from the same source – namely the role of abstract models of computation in Computer Science. Until Alan Turing's seminal work in 1936, computation was an activity of the human mind; much like love and peace. Aspects of the idea that concepts of the human mind can be captured by formal models predate Turing. For example, George Boole's seminal work where the foundations of boolean logic were outlined was titled *An investigation of the Laws of Thought* [Boole, 1854]. However it was Turing who established things on a much grander scale. Turing established the idea that computation has a formal interpretation and that all computation can be captured within a formal system.

Implicit in this achievement however is the fact that abstract models of computation are just that – *abstractions of computation realized in the physical world.* Stated differently, one of the major achievements of Computer Science has been the development of abstract models of computation that shield the discipline from the underlying technology. As effective as these models have been, one must note, however, that they *embody several implicit physical assumptions*. As Tommaso Toffoli explains in his influential 1980 paper:

> Mathematical models of computation are abstract constructions, by their nature unfettered by physical laws. However, if these models are to give indications that are relevant to concrete computing, they must somehow capture, albeit in a selective and stylized way, certain general physical restrictions to which all concrete computing processes are subjected [Toffoli, 1980].

Abstract models of computation, such as Turing machines, λ-calculus and logic gates, allow us to express computation without being concerned about the underlying technology that realizes them in the physical world. Conventional classical models of computation, including boolean logic, the Turing machine, and the λ-calculus, are founded on primitives which correspond to *irreversible* physical processes. For example, a *nand* gate is an irreversible logical operation in the sense that its inputs cannot generally be recovered from observing its output, and so is the operation of overriding a cell on a Turing machine tape with a new symbol, and so is a β-reduction which typically erases or duplicates values in a way that is destructive and irreversible.

The irreversible abstractions we choose as the basis of our most primitive models of computing are at “odds” with the underlying reversible physical reality. The information theoretic underpinnings of physics is best understood through the remarkable *Landauer principle* [Landauer, 1961]. Landauer, through his analysis of Maxwell’s Demon, established the idea that *information is a physical quantity* in much the sense that one would think of an electron as a physical quantity, and that a certain minimum amount of thermodynamic work has to be carried out to erase one bit of information. Experimental verification of the Landauer principle happened in recent years [Bérut et al., 2012]. If information has a physical significance, then it is subject to conservation laws as mass and energy are.

This close connection between information and physics has already led to insights about the entropy of computation and the thermodynamics of confidentiality [Clark et al., 2007, Malacaria, 2007, Malacaria and Smeraldi, 2012]. In *The Thermodynamics of Confidentiality* [Malacaria and Smeraldi, 2012], through a direct application of the Landauer principle, Malacaria establishes the energy dissipation bounds on programs that guarantee confidentiality. However what has been lacking is a basic model of computation that inherently has the right conservation properties and categorical structure. We quote Baez and Stay [2011]:

> Though most physicists don’t know it, quantum physics has long made use of ‘compact symmetric monoidal categories’. Knot theory uses ‘compact braided monoidal categories’, which are slightly more general. However, it became clear in the 1990’s that these more general gadgets are useful in physics too. Logic and computer science used to focus on

> 'cartesian closed categories' – where 'cartesian' can be seen, roughly, as an antonym of 'quantum'. However, thanks to work on linear logic and quantum computation, some logicians and computer scientists have dropped their insistence on cartesianness: now they study more general sorts of 'closed symmetric monoidal categories'.

Jean-Yves Girard introduced *linear logic* in 1987. As he explains [Girard, 1987, pp. 6,17] linear logic embodies a simple and radical change of viewpoint from other logics and this change has a physical flavor. The idea that formal systems must treat computational values and *truth* as 'resources whose usage must be accounted for' was established by linear logic and consequently linear logic is often used as a framework for controlling resource use. Linearity however must not be confused with the criterion of information preservation presented here. Consider $f'(x) = \mathit{if}\ x\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{true}$ which is extensionally equivalent to the constant function $f(x) = \mathit{true}$. In a linear type system that tracks the *syntactic* occurrences of variables, f' would be deemed acceptable because x is linearly used. However the function f is clearly not information-preserving as the value of x is lost. Despite this difference, there does however appear to be some deep connections between linear logic and the physical notions of reversible and quantum computing (see Ch. 9 Sec. 3.1).

2. From Equalities to Computation

What do we mean when we talk about intentional specification of equality leading to an information preserving model of computation? It is useful to take a moment to explain our point of view in simple albeit hand-wavy language, say in terms of say apples and oranges. Paul-André Melliès' monograph on the Categorical Semantics of Linear Logic [Melliès, 2008] contains a wonderful narrative of the idea of proof as interaction in the setting of the *drinker formula*. The proposition $\exists\, x.(P(x) \to \forall\, y.P(y))$ is true in classical logic and is interpreted in the setting of a pub as follows. The formula says that there is a person X in the pub such that if X is drunk then everyone in the pub is drunk. The "proof" of the proposition is interpreted as a conversation (the Indian term *vadh-vivadh* comes to mind here[Sen, 2006]) where the proponent makes a claim that the proposition is true because someone of her choosing appears drunk. The opponent of proposition notices someone (called Y) who is clearly not drunk and points out the sober counter-example. At this point, the proponent changes her witness. She says, "Ah! But, the proposition is still true. I meant to have pointed out Y and if Y is drunk then everyone is drunk". It immediately becomes clear that the proponent has a winning strategy in the argument – for every counterexample provided she can use the counterexample in favor of her position.

The equalities we talk off are processes to transform one thing to the other, in much the sense that the proof above is a strategy to win the argument. For example, if one were to believe that apples and oranges were equal then one would have to show how any given apple can be transformed to an orange i.e. witness of apple-hood get transformed to witnesses of orange-hood (and vice versa). This is inherently different from an extensional notion of equality such as *knowledge is power* and have a more operational sense. Equality in this sense acts more like the the more general notion of *isomorphism* as probably best explained by Barry Mazur's reflections in *When is one thing equal to some other thing?* [Mazur, 2008].

In our formalism, we are not concerned with fruit but with very simple things, namely numbers and sets. When we talk of a set of five apples, we are not concerned about the individuality of the apples that constitute the set. Rather, each apple in the set has the same status as any other apple and we care about the individual apples only to the extent that we can tell them apart to count five distinct ones. We care about the abstract idea of fiveness and its equality with any other abstract idea of fiveness. When used as types, sets take the role of possibility spaces – we are no longer talking about a set of five apples, but talking of one apple out of a possible five. This is also where the switch from sets to information happens: When one apple out of five is identified, it corresponds to the information to discriminate between five choices. Clearly a choice of one out of ten presents more information in the sense that it is a more discriminating choice. When a computation promises an output of a certain type, the resultant value is a witness of one choice out of several. This is the sense in which we are concerned with equalities of sets.

Programs in our model are built from descriptions of equalities or sets/numbers. These are the familiar laws of arithmetic or logic: *Commutativity.* The generalized notion of swapping, i.e. $a \times b = b \times a$ and $a + b = b + a$. *Associativity.* The generalized notion of grouping, i.e. $a \times (b \times c) = (a \times b) \times c$ and $a + (b + c) = (a + b) + c$. *Identity.* Multiplication by 1 and addition with 0, i.e. $a \times 1 = a$ and $a + 0 = a$. *Distributivity.* Multiplication distributes over addition and cancels at zero, i.e. $a \times (b + c) = a \times b + a \times c$ and $a \times 0 = 0$. *Cancellation.* Cancellation is the idea that equal things on both sides of a equality can be canceled out, i.e. $a + b = a + c$ implies $b = c$, and is referred to as a *trace* in category theory [Joyal et al., 1996].

How, one may ask, does this form a computational model? Even in very simple computational models such as the SK combinatory basis, the combinator K provides the basis of choice/conditional by deleting an argument and the combinator S provides the basis for iteration by duplicating an argument. As we will see, distributivity gives us conditionals and trace gives us iteration, albeit in a logically reversible manner. In chapters 3 and 4 we work out the details of the computational model, show that it is Turing complete and develop programs in it including several simple numeric operations, algebraic manipulations of trees and a meta-circular interpreter (see Ch. 5).

More formally, we develop an information preserving model of computation, wherein the process of computing does not gain or lose information. Our model arises from a computational interpretation of type isomorphisms with iso-recursive types and $trace$. This can equivalently be described as computing a commutative semiring with cancellation or as a traced dagger symmetric bi-monoidal category. The category has a groupoid structure and all computations are logically reversible.

We can now describe information preservation in a precise sense. Let 'b' be a (not necessarily finite) type whose values are labeled $b^1, b^2, \ldots$. Let ξ be a random variable of type b that is equal to b^i with probability p_i. The entropy of ξ is defined as $-\sum p_i \log p_i$ [Shannon, 1948]. Consider a function $f : b_1 \rightarrow b_2$ where b_2 is a (not necessarily finite) type whose values are labeled $b_2^1, b_2^2, \ldots$. The output entropy of the function is given by $-\sum q_j \log q_j$ where q_j indicates the probability of the output of the function to have value ${b_2}^j$. We say a function is *information-preserving* if its output entropy is equal to the entropy of its input. See Chapter 6 for details.

Operations that change the information content of programs have a special status. We call them *information effects* and they are encapsulated using an arrow meta-language in much the same way that one would encapsulate other computational effects in the λ-calculus using a monadic metalanguage. The treatment of information effects is analogous to open and closed systems in physics. Closed physical systems conserve mass and energy and are the basic unit of study in physics. Pure computations that do not have information effects are like closed physical systems. They describe equalities. Open systems interact with their environment, possibly exchanging mass or energy. These interactions may be thought of as *effects* that modify the conservation properties of the system. Computations with information effects are much like open systems and they can be converted into pure computations by making explicit the surrounding information environment that they interact with.

Three things follow: (1) We capture the gap between reversible and irreversible computation with a type-and-effect system. (2) Categorically speaking, information effects turn monoidal categories into ones that have a cartesian structure corresponding to conventional computation. (3) We show how conventional irreversible computation such as the λ-calculus can be embedded into this model, such that the embedding makes the implicit information effects of λ-calculus explicit. As a consequence of this approach, many applications in which information manipulation is computationally significant are put within the reach of our conceptual model of computation. Such applications include quantitative information-flow security [Sabelfeld and Myers, 2003], differential privacy [Dwork, 2006], energy-aware computing [Ma et al., 2008, Zeng et al., 2002], VLSI design [Macii and Poncino, 1996], and biochemical models of computation [Cardelli and Zavattaro, 2008].

Chapter 8 focuses on work in progress. It deals with the duality of computation in the setting of information preservation. Andrzej Filinski's work on the symmetric λ-calculus established the fascinating connection that values are dual to continuations, functions are dual to delimited continuations and established the duality of call-by-name and call-by-value evaluation strategies [Filinski, 1989]. We explore Filinski-style duality in the setting of our information preserving model of computing. Here duality does not appear as one

DeMorgan-style dualizing $\neg$ operator, but as two distinct dualizing operations – negation and division. This results in a crisp semantics for negative and fractional types and the resulting system has a equational formalism in zero-totalized fields, aka meadows [Bergstra et al., 2009]. Computational analogues of physical phenomena such as superposition and entanglement appear in the resulting semantics.

3. Technical Overview

The technical development derives from ideas which are implicit in the works of Toffoli [1980], Zuliani [2001] and others [Clark et al., 2007, Ghica, 2007, Malacaria, 2007]. (1) Functions whose output entropy is equal to their input entropy are information preserving functions. (2) A function $f : b_1 \to b_2$ is *logically reversible* if there exists an inverse function $f^\dagger : b_2 \to b_1$ such that $f(v_1) = v_2$ *iff* $f^\dagger(v_2) = v_1$ (where $v_1 : b_1$ and $v_2 : b_2$) (3) Logically reversible functions are information preserving.

The main technical contributions of this thesis build on the observations above and have been published at *Reversible Computation 2011* [Bowman et al., 2011], *POPL* [James and Sabry, 2012a] and *Reversible Computation 2012* [James and Sabry, 2012b]. These are summarized below:

(1) Based on isomorphisms of sum and product types [Fiore, 2004], we develop a strong normalizing model of computing, which we call Π, that is logically reversible and information preserving. Every computation expressible in Π is an isomorphism.

(2) We extend Π with isorecursive type and $trace$ operators from category theory to obtain Π^o. Every computation expressible in Π^o is a partial isomorphism – i.e. the system admits non-termination. Π^o is Turing complete while being information preserving and logically reversible.

(3) We develop the categorical semantics of these models [Bowman et al., 2011]. We also develop a graphical notation for programming in these models that reminiscent of Penrose diagrams for categories (see [Selinger, 2011] for an excellent survey), Geometry of Interaction (GoI) machines and Proof Nets [Mackie, 1995,

2011]. In the graphical notation, computation is modeled by the flow of particles in a circuit.

(4) Since these new models are substantially different from λ-calculus or familiar high-level languages, a point of concern is how one can effectively develop programs in them. We address this by presenting a straightforward technique for deriving Π^o programs by systematically translating logically reversible small-step abstract machines [James and Sabry, 2012b]. Complex programs can be devised in this way; we demonstrate the derivation of a meta-circular interpreter for Π^o.

(5) We develop an arrow meta-language called ML_{Π^o} (and ML_{Π}) over Π^o (and Π) to encapsulate information effects. This metalanguage serves to encapsulate effects, in much the same way that traditional arrows or monads serve to encapsulate effects over the λ-calculus.

(6) We show how a conventional irreversible model can be expressed in our model. We compile the first-order fragment of typed λ-calculus extended with sums, products and loops to Π^o. This compilation is interesting for two reasons: (1) it exposes the implicit information effects of λ-calculus in the compilation to ML_{Π^o} and (2) the compilation of ML_{Π^o} to Π^o shows that information effects can be erased by treating them as interactions with an explicit information environment.

(7) We develop the notion of 'duality' in Π which gives us not one duality (like in linear logic or Filinski's symmetric λ-calculus [Filinski, 1989]), but two notions of duality – an additive duality and a multiplicative duality. This gives a crisp semantics for negative and fractional types in the context of Π.

Several intriguing areas of exploration remain and the thesis concludes with discussion of some of these, along with other possible applications (see Chapter 9). Of these, the field of reversible and quantum computing is closely related and we survey several connections in depth. The exact connection between Π and linear logic [Girard, 1987] is not fully understood though it is clear that both systems capture related but different notions of resource usage. A closely related area is the connection with duality of computation

[Curien and Herbelin, 2000, Filinski, 1989, Wadler, 2003, Zeilberger, 2010]– i.e. the quest for a unifying framework between ‘computations’, values, continuations and their logical counterparts. The investigation of duality gives rise to negative and fractional types which have an intuitive semantics and are reminiscent of negative information flow, superposition and entanglement from quantum physics [James and Sabry, 2012c].

The most primitive notion of equality that is available to us – that of isomorphisms – already contains the notion of computation. The resulting computational model exposes the fine structure of how computation handles information. We believe that such a model of computing is foundational and we are only beginning to understand its applications and full potential. In conclusion, let me quote Paul Blain Levy’s fine advocacy slogan: Once the fine structure has been exposed, why ignore it?

2 Categorical Preliminaries

This chapter is a collection of category theoretic concepts that are relevant to the development of the rest of the thesis. Most categorical concepts introduced here are limited to their definitions, along with references, and are otherwise accompanied with minimal explanation. This spartan treatment of category theory follows from the fact that this thesis does not use category theory to structure proofs or invent new categorical ideas. However, category theory was an invaluable tool that helped structure and guide the development of ideas in this thesis. In this spirit, we quote Sabry and Wadler [1997]:

> While the step from equalities to reductions seems natural in retrospect, we doubt we would have taken it without the guiding light of category theory. In particular, we were motivated by noting the close resemblance between the notion of equational correspondence in Sabry and Felleisen [1993] and the properties of an adjunction.

In the development of our thesis, category theory was useful primarily in the following way: the fact that our computational model had a clear well-known categorical structure enabled the use of category theory to (1) search for other related mathematical structures and (2) use categorical constructions (such as the 'Int construction') as a source of free algorithms in our computational model.

Excellent general references on the category theory include Barr and Wells [1995], "Conceptual Mathematics" [Lawvere and Schanuel, 1997], the little books by Walters [1992] and Pierce [1991] and the monograph on graphical languages by Selinger [2011]. The definitive category theory text is the aptly named 'Categories for the Working Mathematician' by Mac Lane [1971]. A reader well versed in category theory may peruse the titles and otherwise skip reading this chapter. For readers unfamiliar with the categorical concepts discussed here, we recommend using a text with more explanatory notes. However, the main ideas in this thesis are, we hope, accessible without any background in category theory.

1. Basic Definitions

Definition 1.1 (Category). *A* category *consists of the following data:*

- *Objects (usually denoted as A, B, C...)*
- *Arrows (usually denoted f, g, h...)*

such that

- *For each arrow f there are two objects $dom(f)$ and $cod(f)$ called the domain (source) and codomain (target) of f. If $dom(f)$ is the object A and $cod(f)$ is the object B, then one may write $f : A \to B$.*
- *Given arrows $f : A \to B$ and $g : B \to C$ there exists the arrow $g \circ f : A \to C$ called the composite of f and g.*
- *For each object A there is an arrow $id_A : A \to A$.*

and the following properties should hold:

- *Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$*
- *Unit: $f \circ id_A = f = id_B \circ f$*

A category is anything that satisfies this specification. To show that some mathematical entity is a category one must state what the objects are, what the arrows are, what arrow composition corresponds to and show that the required properties are satisfied. Arbitrary categories are usually denoted using a cursive font, such as $\mathscr{C}$. The term morphism means the same as arrow and we use these terms interchangeably.

An important concept to keep in mind when reading categorical definitions is that they are to be treated as *specifications*. Halmos [Halmos, 1960] in “Naive Set Theory” explains how to think about ordered pairs such as “(a, b)” in the context of set theory as being possibly realized by the set “{{a}, {a, b}}”. Halmos explains that one should view the later not as the definition of an ordered pair, but as a particular implementation of the specification of ordered pairs. In similar spirit, the category theoretic notion of a function constitutes (1) two sets called the domain and the codomain and (2) for every element x of the domain, identifies an element of the codomain which is denoted $f(x)$. The set-theoretic definition of function that describes it in terms of the subset of the cartesian product of the domain and codomain is only one particular implementation of this specification. Arrows in category theory are a generalization of functions and in the context of specific categories such as **Set** (discussed below) arrows are functions.

Any text on category theory should provide several examples of categories. We briefly mention two common and relevant examples.

(1) The category of sets, **Set**, is the category whose objects are sets and whose arrows are functions. Composition of functions is the definition of composition, $\circ$, in the category. The identity function from S to S is the identity morphism id_S. Further we can check that:

- composition of functions is associative;
- $id_S : S \rightarrow S$ satisfies $f \circ id_S = f$ where $dom(f) = S$ and
- $id_S \circ g = g$ where $cod(g) = S$

There are other categories whose objects are sets, such as the category **Rel** whose arrows are set-theoretic relations, the category **Pfn** whose arrows are partial functions and the category **Fin** whose objects are finite sets and whose arrows are functions.

(2) A typed point-free 'purely functional' programming language L may be considered a category in the following way [Pitt et al., 1986]:

- The types of L are the objects of the category.
- The operations of L are the arrows of the category. The source and target of an arrow are the input and output types of the corresponding functions.

The identity function of a type gives us the required identity arrows. Function composition in the language is the required arrow composition in the category. The required associativity follows from the associativity of function composition in functional languages. Constants, for example $\mathit{True}, \mathit{False} : \mathit{Bool}$, are usually treated as morphisms $\mathit{True} : 1 \rightarrow \mathit{Bool}$ and $\mathit{False} : 1 \rightarrow \mathit{Bool}$ where 1 is the unit/singleton type. The type 1 forms a terminal object (defined below) of the category defined by the functional language L.

An important subtlety of this definition is in the equality of arrows. Indeed the programs f and $f\ o\ id$ are different syntactic programs and their compilation might be different in any concrete implementation of the language L. Hence the equality of arrows should be based upon the semantic equivalence of these two programs, for example, using using *observational equivalence*. The definition of categories allows one to specify the arrows and the objects, but is not parametrized by an equivalence relationship over the arrows. The '$=$' in the definition of categories is the mathematical equivalence of the mathematical entities that are the arrows. Hence the 'operations of L' that are the arrows must already include the relevant equivalence relation on L programs. This is sometimes referred to as a *term model* and essentially states that operations of the language L are the equivalence classes of the functions/programs one writes in L. Thus f and $f \circ id$ become *the same arrow*. See Section 2.2 of Barr and Wells [Barr and Wells, 1995] for a detailed discussion and Definition 2.4 of Moggi [Moggi, 1991] for an example of such usage.

Definition 1.2 (Hom sets)**.** *For objects A and B of a category, the set of all arrows with source A and target B is denoted $Hom(A, B)$ and is called a* Hom set*.*

Definition 1.3 (Initial Objects)**.** *An object T of a category $\mathscr{C}$ is called an initial object if there is exactly one arrow $T \to A$ for each object A of $\mathscr{C}$.*

Definition 1.4 (Terminal Objects)**.** *An object T of a category $\mathscr{C}$ is called terminal if there is exactly one arrow $A \to T$ for each object A of $\mathscr{C}$.*

Usually terminal objects are denoted by the special object name '1' and initial objects with the special name '0'.

Definition 1.5 (Isomorphism)**.** *Two objects A and B are isomorphic if there exist arrows $f : A \to B$ and $g : B \to A$ such that $f\ o\ g : B \to B$ is the identity arrow of B and $g \circ f : A \to A$ is the identity arrow of A.*

The arrows f and g are referred to as isomorphisms and each may be described as the inverse of the other. Note that the definition of isomorphism is implicitly transitive, i.e. if objects A and B are isomorphic and objects B and C are isomorphic, then A and C are isomorphic. All terminal objects of a category are isomorphic to each other. Similarly, all initial objects of a category are isomorphic to each other. In the category $\mathbf{Set}$ bijective functions are isomorphisms. In later chapters we will use the (somewhat *evil* [nLab, 2012]) notion of *partial isomorphisms* to mean maps whose composition is not always defined, but behave like identity when defined.

Definition 1.6 (Functor)**.** *For categories $\mathscr{C}$ and $\mathscr{D}$, a functor $F : \mathscr{C} \to \mathscr{D}$ is a pair of functions $F_{obj} : \mathscr{C}_{obj} \to \mathscr{D}_{obj}$ and $F_{arr} : \mathscr{C}_{arr} \to \mathscr{D}_{arr}$ which respectively map the objects and arrows of $\mathscr{C}$ to objects and arrows of $\mathscr{D}$, such that:*

- *if $f : A \to B$ in $\mathscr{C}$, then $F_{arr}(f) : F_{obj}(A) \to F_{obj}(B)$ in $\mathscr{D}$.*
- *if A is an object of $\mathscr{C}$, then $F_{arr}(id_A) = id_{F_{obj}(A)}$*
- *$F_{arr}(g \circ f) = F_{arr}(g) \circ F_{arr}(f)$ if $g \circ f$ is defined in $\mathscr{C}$.*

A functor is a structure preserving map between categories. Functors are a central notion of category theory and any text of category theory will contain several examples. Given the notion of functors, one can consider the category of categories which contains categories as objects and functors as the arrows.

Contravariant functors are maps that invert the direction of the arrows. For an arrow $f : A \to B$ in $\mathscr{C}$, a contravariant functor $F : \mathscr{C} \to \mathscr{D}$ would have $F(f) : F(B) \to F(A)$ in $\mathscr{D}$. The direction of composition is also correspondingly reversed, i.e. $F(f \circ g) = F(g) \circ F(f)$.

Definition 1.7 (Dagger Category)**.** *A category $\mathscr{C}$ is a called a dagger category if it has an involutive, identity-on-objects, contravariant functor.*

In essence, this means that for every morphism $f : A \to B$ of $\mathscr{C}$, there exists a morphism $f^\dagger : B \to A$ such that $f^{\dagger\ \dagger} = f$ (involutive), $id_A^\dagger = id_A$ (identity-on-objects) and $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$ (from the fact that † is a contravariant functor). Importantly, what is not specified is how $f^\dagger$ behaves with regard to f. If $f^\dagger$ is the inverse of f, i.e. $f \circ f^\dagger = id$ and $f^\dagger \circ f = id$, then they are called *unitary isomorphisms*. If $f = f^\dagger$ then they are called *self-adjoint morphisms*.

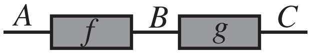
Definition 1.8 (Groupoid)**.** *A groupoid is a category in which every morphism is an isomorphism.*

A group may be thought of as a groupoid with a single object. For a good overview of groupoids see Weinstein [1996] and Higgins [1971].

1.1. Graphical Languages. There are two common graphical languages for categories. The first of these are called *commutation diagrams* and they are a graphical representation of categorical equations. The basic idea is to represent objects of a category as nodes and the morphisms as edges connecting the nodes. The requirement that the arrow $g \circ f$ should be the same as first following the arrow $f : A \to B$ and then following the arrow $g : B \to C$, may be represented as:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[d, "g"] \\
& C
\end{tikzcd}$$

The other style of graphical languages are less widely known and used, but are possibly more relevant to computation. These diagrams go back to Penrose [1971] and are called *string diagrams* or Penrose diagrams. Here one would represent objects as wires and arrows as blocks. Selinger [2011] has an excellent survey of such graphical languages for categories.



This latter style of categorical diagrams have been extended to other categorical concepts such as functors, relatively recently [Melliès, 2006].

2. Monoidal Categories

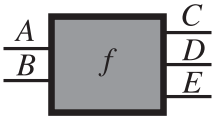
Definition 2.1 (Monoidal Category)**.** *A monoidal category is a category where the objects are equipped with a monoidal structure over the tensor $\otimes$ and the identity object I (usually written as the monoid $(\otimes, I)$).*

- *This extends to an operation on arrows: if $f : A \to C$ and $g : B \to D$, then there exists the arrow $f \otimes g : A \otimes B \to C \otimes D$.*
- *We also have the family of isomorphisms*

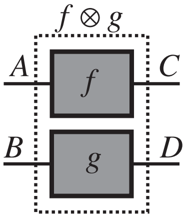
$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$
$$\lambda_A : I \otimes A \to A$$
$$\rho_A : A \otimes I \to A$$

These isomorphisms are subject to several coherence conditions for which we will defer to Selinger's survey paper [Selinger, 2011].

In the string diagrams for categories, monoidal objects allow us to model blocks with multiple inputs and outputs. For instance $f : A \otimes B \to C \otimes D \otimes E$ may be represented as:



The parallel composition of arrows $f \otimes g$ can be represented as the parallel composition of blocks:



Definition 2.2 (Symmetric Monoidal Category)**.** *A symmetric monoidal category is a monoidal category equipped with a natural family of isomorphisms that are self inverses denoted by*

$swap_{A,B} : A \otimes B \to B \otimes A$.

These are further subject to coherence conditions which we have skipped, but may be found in Selinger's survey paper.

Definition 2.3 (Dagger Symmetric Monoidal Category)**.** *A dagger symmetric monoidal category is a symmetric monoidal category that is equipped with a dagger structure – i.e. there is a involutive, identity-on-objects, contravariant functor that provides an adjoint arrow* $f^{\dagger} : B \to A$ *for every arrow* $f : A \to B$*, with the usual composition rules.*

Definition 2.4 (Bimonoidal Category)**.** *A bimonoidal category is a category equipped with two monoidal tensors* $\otimes$ *(with identity object* 1*) and* $+$ *(with identity object* 0*), such that* $\otimes$ *distributes over* $+$*:*

(1) $A \otimes (B + C)$ *is naturally isomorphic to* $(A \otimes B) + (A \otimes C)$*.*
(2) $A \otimes 0$ *is naturally isomorphic to* 0*.*

The bimonoidal structure parallels that of addition and multiplication in semirings. The standard example of a bimonoidal category is **Set** where the tensor operations are cartesian product and disjoint union respectively.

3. Traces, Dual Objects and Exponentials

Definition 3.1 (Traced Monoidal Category)**.** *A traced monoidal category is a category that is equipped with a family of operations*

$$trace_X : Hom(X \otimes A, X \otimes B) \to Hom(A, B).$$

such that the following axioms hold:

- *Tightening*

  $$trace_X((id_X \otimes g) \circ f \circ (id_X \otimes h)) = g \circ trace_X(f) \circ h$$

- *Sliding*

  $$trace_Y(f \circ (g \otimes id_A)) = trace_X((g \otimes id_B) \circ f)$$

  where

  $$f : X \otimes A \to Y \otimes B$$

  $$g : Y \to X$$

- *Vanishing*
  - $trace_I(id_I \otimes f) = f$
  - $trace_{X \otimes Y}(f) = trace_Y(trace_X(f))$

- *Strengthening*

  $trace_X(f \otimes g) = trace_X(f) \otimes g$

In other words, in a traced monoidal category for any morphism $f : X \otimes A \to X \otimes B$ we can construct a morphism $trace(f) : A \to B$ such that the required axioms are satisfied. Some clarification need to be made:

- The above definition assumes that the monoidal category is 'strict' i.e. that association operations are implicit.
- The above definition is called a 'left' trace to distinguish from a 'right' trace, which corresponds to the operation:

  $trace_X : Hom(A \otimes X, B \otimes X) \to Hom(A, B)$

  We are not concerned with this distinction since we will be working with symmetric monoidal categories.

Traced monoidal categories were introduced by Joyal et al. [1996] as the categorical basis for cyclic structures in mathematics (such as knots). In computer science traced monoidal categories have been studied as a semantic basis for recursion, fixpoints and feedback loops [Abramsky, 2005a, Bloom and Ésik, 1993, Hasegawa, 1997]. The required coherence conditions for categorical traces are best understood when viewed graphically and both Selinger [2011] and Hasegawa [1997] are good sources.

Definition 3.2 (Cartesian Closed Category)**.** *A cartesian closed category $\mathscr{C}$ satisfies the following requirements:*

- *$\mathscr{C}$ has a terminal object.*
- *For each pair of objects A and B, $\mathscr{C}$ has a product object $A \times B$ with projections $p_1 : A \times B \to A$ and $p_2 : A \times B \to B$.*
- *For every pair of objects A and B there is an object $[A \to B]$ and an arrow $eval : [A \to B] \times A \to B$ such that for any $f : C \times A \to B$ there is a unique arrow $f' : C \to [A \to B]$ such that:*

  $eval \circ (f' \times id_A) = f$

Note that a cartesian closed category, usually referred to as a CCC, is defined in terms of products that have projection morphisms, and not in terms of tensors. CCCs are used in the categorical semantics of typed λ-calculi with the objects $[A \to B]$, usually called exponential objects, representing first class functions. From the perspective of λ-calculi, the arrow f' may be understood as generating a curried form of the function denoted by the arrow f.

3

Finite Types and their Isomorphisms

This chapter introduces a strongly normalizing model of computation that we call Π. The computational model is derived from the notion of isomorphisms between finite types and serves as the basis for all of our subsequent development. The computational model is logically reversible in the sense that for every computation (expressible in Π) that transforms inputs into outputs, one can derive a computation that transforms the outputs back into the inputs. In fact, forward and backward execution have equal status in Π and execution in one direction is not any more special than execution in the other direction.

Consequently computations expressible in Π are also information preserving. The intuitive way to understand this is that the transformation from inputs to outputs cannot lose any information because, if they do, the inputs would not be constructable from the outputs. Finally, the computational model is universal for boolean circuits i.e. every finite boolean circuit using any of the conventional 'logic gates' can be expressed in this system. Hence, even though Π is not Turing complete, it can express a large class of interesting computations.

While we provide a term language for such an information preserving model of computation, write programs in it and develop its theory, the idea that one can compute in an information preserving manner goes back at least to Fredkin, Landauer, Bennett and Toffoli [Bennett, 2003, Bennett and Landauer, 1985, Bennett, 1973, Fredkin and Toffoli, 1982, Toffoli, 1980]. At its heart, the idea that computation is possible in an information preserving manner appears paradoxical since we compute by taking steps. Each step changes the state of the system thereby raising the question whether information has somehow been lost or gained by taking that step. This view is paralyzing because it would appear that no computational steps are possible at all and hence no computation is possible. The quote below is by Marvin Minsky from the book *Feynman and Cellular Vacuum, Feynman and Computation: Exploring the Limits of Computers.*

> Ed Fredkin pursued the idea that information must be finite in density.
> One day, he announced that things must be even more simple than that.
> He said that he was going to assume that information itself is conserved.

> "You're out of you mind, Ed." I pronounced. "That's completely ridiculous. Nothing could happen in such a world. There couldn't even be logical gates. No decisions could ever be made." But when Fredkin gets one of his ideas, he's quite immune to objections like that; indeed, they fuel him with energy. Soon he went on to assume that information processing must also be reversible — and invented what's now called the Fredkin gate.
>
> When Bennett and Landauer wrote their paper showing that reversible, non-dissipative computation was indeed possible in principle, I was asked to be a referee for the journal they had submitted it to. I read the paper over and over — every day for a solid month. Finally I sent a note to the journal. "This result just doesn't seem possible. However, I have read it very carefully, and cannot find where might be the mistake. I suppose you'll just have to publish it!"
>
> Feynman too was skeptical. However, instead of merely checking their proof, he set out to prove it for himself. He came up with a different and simpler proof, which convinced many others that the discovery was believable. Soon he started to design the first quantum computers.

We proceed slowly by first defining our choice of types and type isomorphisms and then proceeding to build a computational model based on these. In later chapters we will make the notion of information preservation more precise.

1. Finite Types

1.1. Finite Types and their Values.

Definition 1.1. *Finite Types, b: We define the syntactic category of finite types/base types, b, in the standard way.*

$$\textit{Base types}, b \quad = \quad 0 \mid 1 \mid b + b \mid b \times b$$

Definition 1.2. *Values, v: The syntactic category of values, v, is defined in the standard way.*

$$Values, v \quad ::= \quad () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v)$$

Each finite type b is inhabited by a finite number of values. The set of finite types b is constructed using sums and products from the primitive types 0 and 1. The type 0 has no inhabitants. The type 1 has exactly one inhabitant called "unit" and denoted by $()$.

Sum types have the form $b_1 + b_2$ and represent the disjoint union of the types b_1 and b_2. Values of the sum type can be distinguished using their left and right constructors. Pairs of the form $b_1 \times b_2$ allow the encoding of tuples where the first component is of type b_1 and the second component has type b_2. The formal relationship between values and their types is given by the type system below:

Definition 1.3. *Type System. Sequents of this type system have the form $\vdash v : b$. There is no derivation for the type 0 which has no inhabitants.*

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

Example. Here is a short discussion for readers who may be unfamiliar with sum and product types. Let us say we have the numbers 5 and 10 as having the type int, booleans true and false as having the type bool and strings "hello" and "world" having the type string.

The product type $\mathit{int} \times \mathit{string}$ is to be read as having both a value of type int and a value of type string. Thus (5,"hello"), (10, "hello") and (5, "world") have the type $\mathit{int} \times \mathit{string}$. The value $("\mathit{hello}", 5)$ does not have this type – it has the type $\mathit{string} \times \mathit{int}$. The types $\mathit{int} \times \mathit{string}$ and $\mathit{string} \times \mathit{int}$ are not equal, but are equivalent in a certain sense – this notion of isomorphism of types is formalized in the next section.

The sum type $\mathit{int} + \mathit{string}$ says that we may either have a value of type int or a value of type string. The value will be tagged to identify if it is a value of the type on the left or the right of the $+$. Thus left 5, left 10, right "hello" and right "world" are values that have type $\mathit{int} + \mathit{string}$. The value left "hello" does not have type $\mathit{int} + \mathit{string}$, but it can have type

$string + int$. In general however one cannot look at one value and determine the type of the value – the value $left$ 5 may have the type $int + string$ and also the type $int + bool$.

Types whose values can be placed in one to one correspondence are considered isomorphic. For instance the type $bool + bool$ which has values $left\ true$, $left\ false$, $right\ false$ and $right\ true$ is isomorphic to the type $bool \times bool$ which has values $(true, true)$, $(false, true)$, $(true, false)$ and $(false, false)$. One possible mapping is:

$left\ true$	$(true, true)$
$left\ false$	$(false, true)$
$right\ true$	$(true, false)$
$right\ false$	$(false, false)$

While the types $string$ and nat have an infinite number of values, here we are concerned with types that only have a finite number of values, i.e. we are concerned with types that have one value, two values, three values etc. Instead of including types like $bool$, fixed precision int etc. natively to our type system, we include the primitive types 0 and 1 to our system, the idea being that every other finite number can be expressed with $+$ and $\times$. Hence the syntax of our types is given by the minimal syntax of base types, b. More details may be found in excellent references such as “TAPL” [Pierce, 2002].

Definition 1.4. *Least Value of a Type. The least value of a type* b*, denoted* $\phi(b)$*, is defined to be:*

$$\begin{aligned}
\phi(0) &= undefined \\
\phi(1) &= () \\
\phi(b_1 + b_2) &= left\ \phi(b_1) \\
\phi(b_1 \times b_2) &= (\phi(b_1), \phi(b_2))
\end{aligned}$$

Not every type has a least value. This definition must be used with care in situations where it can be shown that that we do not require the least of 0.

Definition 1.5. *Size of a type. The size of type* b *is denoted by* $|b|$ *and is defined to be:*

$$\begin{array}{rcl} |\,0\,| & = & 0 \\ |\,1\,| & = & 1 \\ |\,b_1 + b_2\,| & = & |\,b_1\,| \;+\; |\,b_2\,| \\ |\,b_1 \times b_2\,| & = & |\,b_1\,| \;\times\; |\,b_2\,| \end{array}$$

In the above definition the 0, 1, $+$ and $\times$ that occur on the right side of the $=$ are arithmetic numbers and operations. Thus the size of $(1 + 1) \times (1 + 0)$ is 2. The 'size' of a type may be understood as the number of values that inhabit the type. While this interpretation is valid for now, in Chapter 8 we will see extensions to the type system that make this interpretation tenuous.

1.2. Isomorphisms of Finite Types. Two types b_1 and b_2 are isomorphic, written as $b_1 \leftrightarrow b_2$, if we can construct a bijective map between their values. Every finite type can be identified, upto isomorphism, by a natural number representing the number of elements of the type.

The sound and complete set of isomorphisms between finite types is given by the *congruence closure* of a set of base isomorphisms as defined below [Fiore, 2004]:

Definition 1.6. *Base isomorphisms for finite types.*

$$\begin{array}{rclr} 0 + b & \leftrightarrow & b & \textit{identity for } + \\ b_1 + b_2 & \leftrightarrow & b_2 + b_1 & \textit{commutativity for } + \\ b_1 + (b_2 + b_3) & \leftrightarrow & (b_1 + b_2) + b_3 & \textit{associativity for } + \\ \\ 1 \times b & \leftrightarrow & b & \textit{identity for } \times \\ b_1 \times b_2 & \leftrightarrow & b_2 \times b_1 & \textit{commutativity for } \times \\ b_1 \times (b_2 \times b_3) & \leftrightarrow & (b_1 \times b_2) \times b_3 & \textit{associativity for } \times \\ \\ 0 \times b & \leftrightarrow & 0 & \textit{distribute } \times \textit{ over } 0 \\ (b_1 + b_2) \times b_3 & \leftrightarrow & (b_1 \times b_3) + (b_2 \times b_3) & \textit{distribute } \times \textit{ over } + \end{array}$$

These isomorphisms are already familiar to us from arithmetic or logic (reading 0 as $false$, 1 as $true$, $\times$ as conjunction, and $+$ as disjunction).

Definition 1.7. *Congruence Closure. The reflexive, symmetric, transitive, compatible closure, i.e. the congruence closure, over the base isomorphisms is defined to be:*

$$\frac{}{b \leftrightarrow b} \quad \frac{b_1 \leftrightarrow b_2}{b_2 \leftrightarrow b_1} \quad \frac{b_1 \leftrightarrow b_2 \quad b_2 \leftrightarrow b_3}{b_1 \leftrightarrow b_3}$$

$$\frac{b_1 \leftrightarrow b_3 \quad b_2 \leftrightarrow b_4}{(b_1 + b_2) \leftrightarrow (b_3 + b_4)} \quad \frac{b_1 \leftrightarrow b_3 \quad b_2 \leftrightarrow b_4}{(b_1 \times b_2) \leftrightarrow (b_3 \times b_4)}$$

Despite the similarities to logical axioms, note however that the isomorphisms do not include some familiar logical tautologies, in particular:

$$\begin{array}{rcl} b \times b & \not\leftrightarrow & b \\ b_1 + (b_2 \times b_3) & \not\leftrightarrow & (b_1 + b_2) \times (b_1 + b_3) \end{array}$$

Even though these identities are expected in propositional logic, they are not satisfied in standard arithmetic nor in any logic that accounts for resources like linear logic.

Definition 1.8. *Canonical Type. For any given size n, we define the canonical type of that size, b^n, to be:*

$$\begin{array}{rcl} b^0 & = & 0 \\ b^1 & = & 1 \\ b^{n+1} & = & 1 + b_n \end{array}$$

2. A Computational Model from Type Isomorphisms: Π

We are interested in studying the computational content of type isomorphisms. This section derives a programming language, Π, from the type isomorphisms discussed in Section 1.2. The defining feature of Π is that every expressible computation is an isomorphism and will have a type of the form $b_1 \leftrightarrow b_2$. [1]

[1] The name Π originated from the view that Π is a term language for permutations and in mathematics the symbol π is often used to denote permutations.

Definition 2.1. *Primitive operators of* Π

$$\begin{array}{rrcll}
\mathit{zeroe}: & 0+b & \leftrightarrow & b & :\mathit{zeroi} \\
\mathit{swap}^{+}: & b_1+b_2 & \leftrightarrow & b_2+b_1 & :\mathit{swap}^{+} \\
\mathit{assocl}^{+}: & b_1+(b_2+b_3) & \leftrightarrow & (b_1+b_2)+b_3 & :\mathit{assocr}^{+} \\
\mathit{unite}: & 1\times b & \leftrightarrow & b & :\mathit{uniti} \\
\mathit{swap}^{\times}: & b_1\times b_2 & \leftrightarrow & b_2\times b_1 & :\mathit{swap}^{\times} \\
\mathit{assocl}^{\times}: & b_1\times(b_2\times b_3) & \leftrightarrow & (b_1\times b_2)\times b_3 & :\mathit{assocr}^{\times} \\
\mathit{distrib}_0: & 0\times b & \leftrightarrow & 0 & :\mathit{factor}_0 \\
\mathit{distrib}: & (b_1+b_2)\times b_3 & \leftrightarrow & (b_1\times b_3)+(b_2\times b_3) & :\mathit{factor}
\end{array}$$

Each line of this table is to be read as the definition of one or two operators. For example, corresponding to the *identity of* $\times$ isomorphism ($1 \times b \leftrightarrow b$) we have the two operators $\mathit{unite} : 1\times b \leftrightarrow b$ (reading the isomorphism from left to right) and $\mathit{uniti} : b \leftrightarrow 1\times b$ (reading the isomorphism from right to left). These operators are inverses of each other. Each of the two cases of commutativity defines one operator that is its own inverse.

Having defined primitive operators, we need some means of composing them. We construct the composition combinators out of the closure conditions for isomorphisms.

Definition 2.2. *Composition in* Π

$$\frac{}{\mathit{id} : b \leftrightarrow b} \qquad \frac{c : b_1 \leftrightarrow b_2}{\mathit{sym}\ c : b_2 \leftrightarrow b_1} \qquad \frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3}$$

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

Thus we have program constructs that witness reflexivity id, symmetry sym, and transitivity $\fatsemi$ and two parallel composition combinators, one for sums $+$ and one for pairs $\times$.

Definition 2.3. *(Syntax of Π) We collect our types, values, and combinators, to get the full language definition.*

$$\begin{array}{rcl}
\mathit{value\ types}, b & ::= & 0 \mid 1 \mid b + b \mid b \times b \\
\mathit{values}, v & ::= & () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v)
\end{array}$$

$$\begin{array}{rcl}
\mathit{combinator\ types}, t & ::= & b \leftrightarrow b \\
\mathit{isomorphisms}, \mathit{iso} & ::= & \mathit{zeroe} \mid \mathit{zeroi} \mid \mathit{swap}^{+} \mid \mathit{assocl}^{+} \mid \mathit{assocr}^{+} \\
& \mid & \mathit{unite} \mid \mathit{uniti} \mid \mathit{swap}^{\times} \mid \mathit{assocl}^{\times} \mid \mathit{assocr}^{\times} \\
& \mid & \mathit{distrib}_0 \mid \mathit{factor}_0 \mid \mathit{distrib} \mid \mathit{factor} \\
\mathit{combinators}, c & ::= & \mathit{iso} \mid \mathit{id} \mid \mathit{sym}\ c \mid c \fatsemi c \mid c + c \mid c \times c
\end{array}$$

Definition 2.4. *Operational Semantics for Π. 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 $c\ v_1 \mapsto v_2$ are given below.*

Additive fragment:

$$\begin{array}{rlcl}
\mathit{zeroe} & (\mathit{right}\ v) & \mapsto & v \\
\mathit{zeroi} & v & \mapsto & \mathit{right}\ v \\
\mathit{swap}^{+} & (\mathit{left}\ v) & \mapsto & \mathit{right}\ v \\
\mathit{swap}^{+} & (\mathit{right}\ v) & \mapsto & \mathit{left}\ v \\
\mathit{assocl}^{+} & (\mathit{left}\ v_1) & \mapsto & \mathit{left}\ (\mathit{left}\ v_1) \\
\mathit{assocl}^{+} & (\mathit{right}\ (\mathit{left}\ v_2)) & \mapsto & \mathit{left}\ (\mathit{right}\ v_2) \\
\mathit{assocl}^{+} & (\mathit{right}\ (\mathit{right}\ v_3)) & \mapsto & \mathit{right}\ v_3 \\
\mathit{assocr}^{+} & (\mathit{left}\ (\mathit{left}\ v_1)) & \mapsto & \mathit{left}\ v_1 \\
\mathit{assocr}^{+} & (\mathit{left}\ (\mathit{right}\ v_2)) & \mapsto & \mathit{right}\ (\mathit{left}\ v_2) \\
\mathit{assocr}^{+} & (\mathit{right}\ v_3) & \mapsto & \mathit{right}\ (\mathit{right}\ v_3)
\end{array}$$

Multiplicative fragment:

$$\begin{array}{rlcl}
\mathit{unite} & ((), v) & \mapsto & v \\
\mathit{uniti} & v & \mapsto & ((), v) \\
\mathit{swap}^{\times} & (v_1, v_2) & \mapsto & (v_2, v_1) \\
\mathit{assocl}^{\times} & (v_1, (v_2, v_3)) & \mapsto & ((v_1, v_2), v_3) \\
\mathit{assocr}^{\times} & ((v_1, v_2), v_3) & \mapsto & (v_1, (v_2, v_3))
\end{array}$$

Distributivity and factoring:

$$\begin{array}{rlcl}
\mathit{distrib} & (\mathit{left}\ v_1, v_3) & \mapsto & \mathit{left}\ (v_1, v_3) \\
\mathit{distrib} & (\mathit{right}\ v_2, v_3) & \mapsto & \mathit{right}\ (v_2, v_3) \\
\mathit{factor} & (\mathit{left}\ (v_1, v_3)) & \mapsto & (\mathit{left}\ v_1, v_3) \\
\mathit{factor} & (\mathit{right}\ (v_2, v_3)) & \mapsto & (\mathit{right}\ v_2, v_3)
\end{array}$$

The evaluation rules of the composition combinators are given below:

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

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

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

Since there are no values that have the type 0, the reductions for the combinators zeroe, zeroi, $\mathit{distrib}_0$ and factor_0 omit the impossible cases. Also, the semantics of $\mathit{sym}\ c$ are defined in terms of the adjoint of c (denoted by $c^{\dagger}$), which we formalize below.

Definition 2.5 (Adjoints, $c^{\dagger}$)**.** *The adjoint of a combinator c is defined by $c^{\dagger}$ as follows:*

- *For primitive isomophisms c, $c^{\dagger}$ is given by its isomorphic inverse operator from the table in Definition 2.1.*
- $(c_1 \times c_2)^{\dagger} = c_1^{\dagger} \times c_2^{\dagger}$
- $(c_1 + c_2)^{\dagger} = c_1^{\dagger} + c_2^{\dagger}$
- $(c_1 \fatsemi c_2)^{\dagger} = c_2^{\dagger} \fatsemi c_1^{\dagger}$. *(Note that the order of combinators has been reversed).*

Definition 2.6 (Observational Equivalence. $c_1 = c_2$)**.** *Combinators c_1 and c_2, where $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_1 \leftrightarrow b_2$, are observationally equivalent,$c_1 = c_2$, iff*

$\forall\, v_1 : b_1, v_2 : b_2.\ c_1\ v_1 \mapsto v_2 \Longleftrightarrow c_2\ v_1 \mapsto v_2.$

Proposition 2.7. *Canonical Isomorphism. For any type b there exists the isomorphism $b \leftrightarrow b^{|b|}$.*

Proof. The fact that such an isomorphism exists is evident from the definition of size and what it means for two types to be isomorphic. While many equivalent constructions are possible for any type b, one such construction is given by $\llbracket b \rrbracket$:

$$
\begin{array}{rcl}
\llbracket 0 \rrbracket & = & id \\
\llbracket 1 \rrbracket & = & id \\
\llbracket 1 + b \rrbracket & = & id + \llbracket b \rrbracket \\
\llbracket (b_1 + b_2) + b_3 \rrbracket & = & assocr^{+} \fatsemi \llbracket b_1 + (b_2 + b_3) \rrbracket \\
\llbracket b_1 + b_2 \rrbracket & = & (\llbracket b_1 \rrbracket + id) \fatsemi \llbracket b_1^{|b_1|} + b_2 \rrbracket \\
\llbracket (b_1 \times b_2) \times b_3 \rrbracket & = & assocr^{\times} \fatsemi \llbracket b_1 \times (b_2 \times b_3) \rrbracket \\
\llbracket (b_1 + b_2) \times b_3 \rrbracket & = & distrib \fatsemi \llbracket b_1 \times b_3 + b_2 \times b_3 \rrbracket
\end{array}
$$

□

2.1. Logical Reversibility. The idea of logical reversibility is explored in the writings of Zuliani [2001], Fredkin and Toffoli [1982] and Bennett [1973] in the context of *reversible computation*. We use a specific definition for logical reversibility in this thesis.

A computational model is said to be *logically reversible* if and only if every expressible computation $f : b_1 \rightarrow b_2$ has an inverse $g : b_2 \rightarrow b_1$ such that for all $v_1 : b_1$ and $v_2 : b_2$, i.e. $f(v_1) = v_2$ if and only if $g(v_2) = v_1$. The important thing to note is that this definition holds in the presence of non-termination: if there is no v_2 such that $f(v_1) = v_2$, logical reversibility still holds. Further, this definition is symmetric in f and g in that both computations have equal status in the computational model – i.e. it is not the case that users of the computational model can specify computation only in one direction and the reverse executions are implicit or derivable in a way that is internal to the system.

These ideas extend to Π. By design, every program construct in Π has an adjoint that works in the other direction.

Proposition 2.8. *Each combinator $c : b_1 \leftrightarrow b_2$ has an adjoint $c^{\dagger} : b_2 \leftrightarrow b_1$.*

PROOF. The proof follows by induction on the structure of derivation of $c : b_1 \leftrightarrow b_2$:

- For primitive isomorphisms by definition $c : b_1 \leftrightarrow b_2$ and $c^\dagger : b_2 \leftrightarrow b_1$.
- For $c_1 + c_2 : (b_1 + b_3) \leftrightarrow (b_2 + b_4)$, by induction
  - $c_1 : b_1 \leftrightarrow b_2$ and $c_1^\dagger : b_2 \leftrightarrow b_1$
  - $c_2 : b_3 \leftrightarrow b_4$ and $c_2^\dagger : b_4 \leftrightarrow b_3$

  Thus $c_1^\dagger + c_2^\dagger : (b_2 + b_4) \leftrightarrow (b_1 + b_3)$.
- The case for $c_1 \times c_2$ is similar.
- For $c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3$, by induction
  - $c_1 : b_1 \leftrightarrow b_2$ and $c_1^\dagger : b_2 \leftrightarrow b_1$
  - $c_2 : b_2 \leftrightarrow b_3$ and $c_2^\dagger : b_3 \leftrightarrow b_2$

  Thus $c_2^\dagger \fatsemi c_1^\dagger : b_3 \leftrightarrow b_1$.

□

The use of the *sym* constructor uses the adjoint to reverse the program. We can now formalize that the adjoint of each construct c is its inverse in the sense that the evaluation of the adjoint maps the output of c to its input. This property is a strong version of logical reversibility in which the inverse of a program is simply obtained by the adjoint operation.

Proposition 2.9 (Logical Reversibility). *$c\ v \mapsto v'$ iff $c^\dagger v' \mapsto v$.*

PROOF. Proof follows from induction of the derivation of $c\ v \mapsto v'$

- For primitive isomorphisms $c\ v \mapsto v'$, we know that $c^\dagger\ v' \mapsto v$ by examining the operational definitions, 2.4.
- For $(c_1 \times c_2)\ v \mapsto v'$, the derivation must have the form

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

where $v = (v_1, v_2)$ and $v' = (v_3, v_4)$. By induction we have $c_1^\dagger\ v_3 \mapsto v_1$ and $c_2^\dagger\ v_4 \mapsto v_2$ and thus we have:

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

- The case for $(c_1 + c_2)$ works out similar to the above with the notable difference that we have to examine the derivation taking the *left* and *right* branches, each of which readily follows by induction.
- The case for $c_1 \fatsemi c_2$ has the derivation

$$\frac{c_1\ v_1 \mapsto v \quad c_2\ v \mapsto v_2}{(c_1 \fatsemi c_2)\ v_1 \mapsto v_2}$$

and thus by induction we have $c_1^{\dagger} : v \mapsto v_1$ and $c_2^{\dagger}\ v_2 \mapsto v$. Hence

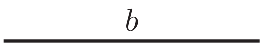
$$\frac{c_2^{\dagger}\ v_2 \mapsto v \quad c_1^{\dagger}\ v \mapsto v_1}{(c_2^{\dagger} \fatsemi c_1^{\dagger})\ v_2 \mapsto v_1}$$

□

3. A Graphical Language for Π

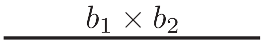
Combinators of Π can be written in terms of the operators described in definition 2.3 or via a graphical language similar in spirit to those developed for Geometry of Interaction [Mackie, 1995] and string diagrams for category theory [Selinger, 2011]. Modulo some conventions and shorthand we describe here, the wiring diagrams are equivalent to the operator based (syntactic) description of programs. Π combinators expressed in this graphical language look like "wiring diagrams". Values take the form of "particles" that flow along the wires. Computation is expressed by the flow of particles.

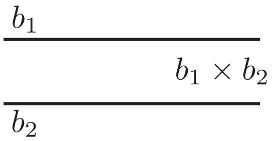
- The simplest sort of diagram is the $id : b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type b. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted.



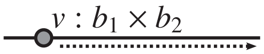
Values flow from left to right in the graphical language of Π. When tracing a computation, one might imagine a value v of type b on the wire, as shown below.

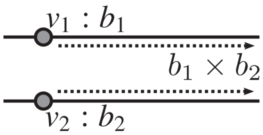
- The product type $b_1 \times b_2$ may be represented both as one wire labeled $b_1 \times b_2$ or by two parallel wires labeled b_1 and b_2. Both representations may be used interchangeably.



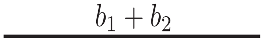


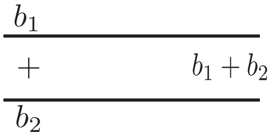
When tracing execution using particles, one should think of one particle on each wire or alternatively as in folklore in the literature on monoidal categories as a "wave."



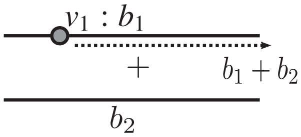


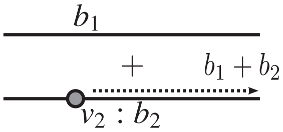
- Sum types may similarly be represented using using parallel wires with a $+$ operator between them.



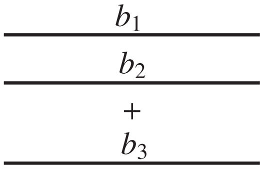


When tracing the execution of $b_1 + b_2$ represented by one wire, one can think of a value of the form *left* v_1 or *right* v_2 as flowing on the wire, where $v_1 : b_1$ and $v_2 : b_2$. When tracing the execution of two additive wires, a value can reside on only one of the two wires.

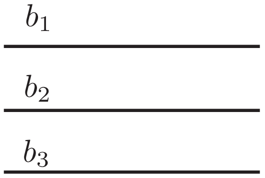




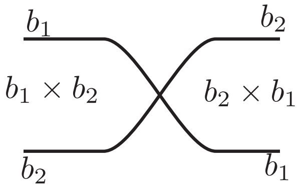
- When representing complex types like $(b_1 \times b_2) + b_3$ some visual grouping of the wires may be done to aid readability. The exact type however will always be clarified by the context of the diagram.

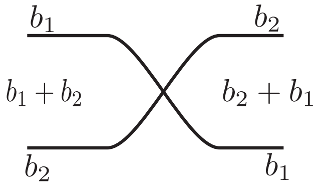


- Associativity is entirely skipped in the graphical language. Hence three parallel wires may be inferred as $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$, based on the context. This is much like handling of associativity in the graphical representations of categories.

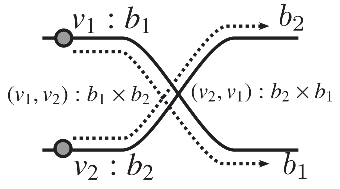


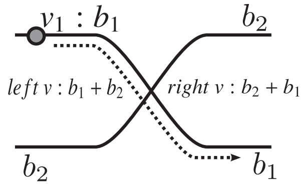
- Commutativity is represented by crisscrossing wires.



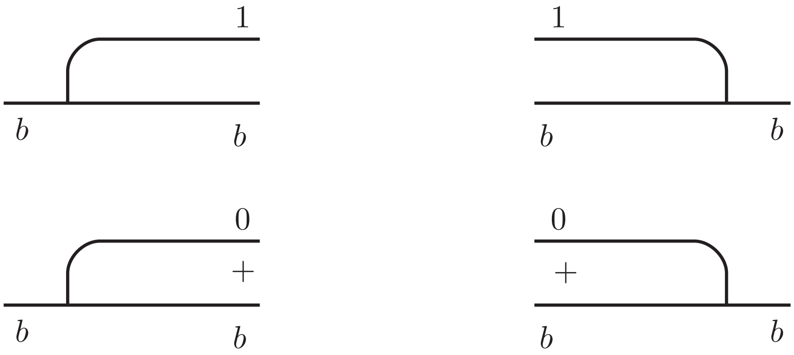


When tracing the execution of $b_1 + b_2$ represented by one wire, one can think of a value of the form *left* v_1 or *right* v_2 as flowing on the wire, where $v_1 : b_1$ and $v_2 : b_2$. By visually tracking the flow of particles on the wires, one can verify that the expected types for commutativity are satisfied.

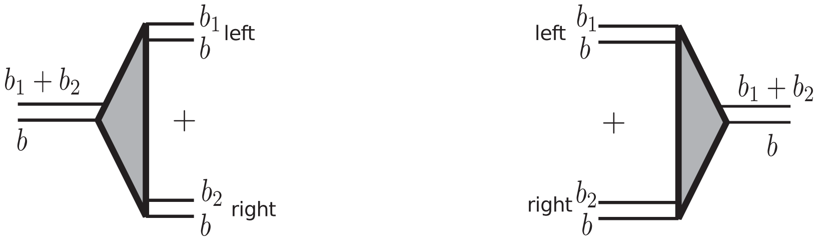




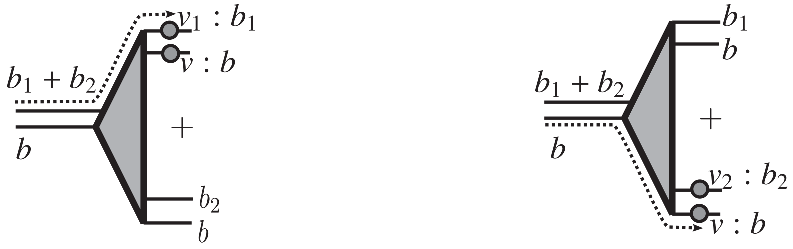
- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, in many diagrams they are omitted and dealt with explicitly only when they are of special interest.



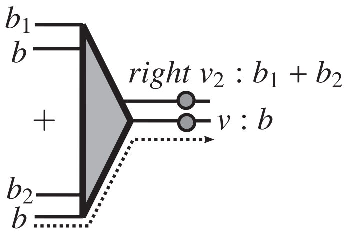
- Distributivity and factoring are represented using the dual boxes shown below:

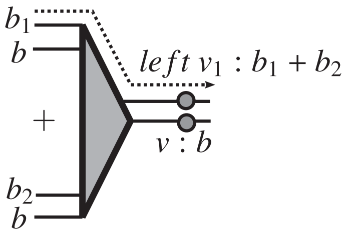


Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type $b_1 + b_2$, as shown below.

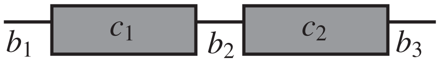


Factoring is the corresponding adjoint operation.

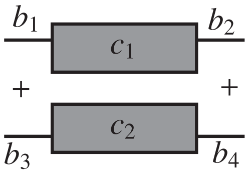


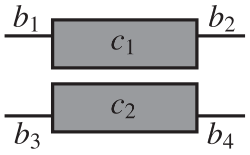


- Combinators can be composed in series ($c_1 \fatsemi c_2$) or parallel. Sequential (series) composition corresponds to connecting the output of one combinator to the input of the next.



There are two forms of parallel composition – combinators can be combined additively $c_1 + c_2$ (shown on the left) or multiplicatively $c_1 \times c_2$ (shown on the right).

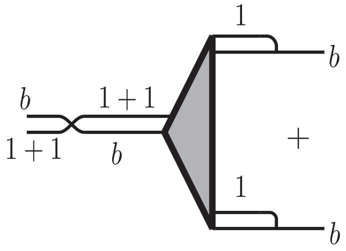




Example. As an example consider the wiring diagram of the combinator c below:

$c : b \times (1 + 1) \leftrightarrow b + b$

$c = swap^{\times} \fatsemi distrib \fatsemi (unite + unite)$



4. Expressing Programs with Π

In this section we introduce programming in Π through several simple examples. We subsequently discuss the expressiveness of the language and discuss various properties.

4.1. Primitive Types.

Booleans. Let us start with encoding booleans. We use the type $1 + 1$ to represent booleans with $\mathit{left}\ ()$ representing true and $\mathit{right}\ ()$ representing false. Boolean negation is straightforward to define:

$\mathit{not} : \mathit{bool} \leftrightarrow \mathit{bool}$

$\mathit{not} = \mathit{swap}^{+}$

It is easy to verify that not changes true to false and vice versa.

Bit Vectors. We can represent n-bit words using an n-ary product of bools. For example, we can represent a 3-bit word, word_3, using the type $\mathit{bool} \times (\mathit{bool} \times \mathit{bool})$. We can perform various operations on these 3-bit words using combinators in Π. For instance the bitwise not operation is the parallel composition of three not operations:

$\mathit{not}_{\mathit{word}_3} :: \mathit{word}_3 \leftrightarrow \mathit{word}_3$

$\mathit{not}_{\mathit{word}_3} = \mathit{not} \times (\mathit{not} \times \mathit{not})$

We can express a 3-bit word reversal operation as follows:

$\mathit{reverse} : \mathit{word}_3 \leftrightarrow \mathit{word}_3$

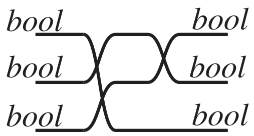
$\mathit{reverse} = \mathit{swap}^{\times} \fatsemi (\mathit{swap}^{\times} \times \mathit{id}) \fatsemi \mathit{assocr}^{\times}$

We can check that $\mathit{reverse}$ does the right thing by applying it to a value $(v_1, (v_2, v_3))$ and writing out the full derivation tree of the reduction. The combinator $\mathit{reverse}$, like many others we will see in this thesis, is formed by sequentially composing several simpler combinators. Instead of presenting the operation of $\mathit{reverse}$ as a derivation tree, it is easier (purely for presentation reasons) to flatten the tree into a sequence of reductions as caused by each component. Such a sequence of reductions is given below:

$$\begin{array}{rl}
 & (v_1, (v_2, v_3)) \\
\mathit{swap}^{\times} & ((v_2, v_3), v_1) \\
\mathit{swap}^{\times} \times \mathit{id} & ((v_3, v_2), v_1) \\
\mathit{assocr}^{\times} & (v_3, (v_2, v_1))
\end{array}$$

On the first line is the initial value. On each subsequent line is a fragment of the $\mathit{reverse}$ combinator and the value that results from applying this combinator to the value on the previous line. For example, $\mathit{swap}^{\times}$ transforms $(v_1,(v_2,v_3))$ to $((v_2,v_3),v_1)$. On the last line we see the expected result with the bits in reverse order.

We can also draw out the graphical representation of the 3-bit reverse combinator. In the graphical representation, it is clear that the combinator achieves the required shuffling.

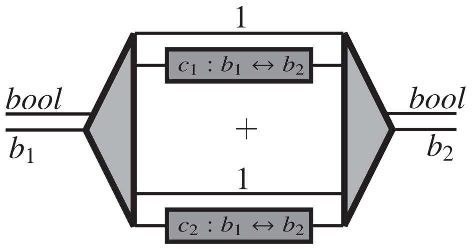


4.2. Expressiveness.

Conditionals. Even though Π lacks conditional expressions, they are expressible using the distributivity and factoring laws. The diagrammatic representation of $\mathit{distrib}$ shows that it redirects the flow of a value $v : b$ based on the value of another one of type $b_1 + b_2$. If we choose $1+1$ to be bool and apply either $c_1 : b \leftrightarrow b$ or $c_2 : b \leftrightarrow b$ to the value v, then we essentially have an 'if' expression.

$$\mathit{if}_{c_1,c_2} : \mathit{bool} \times b \leftrightarrow \mathit{bool} \times b$$

$$\mathit{if}_{c_1,c_2} = \mathit{distrib} \fatsemi ((\mathit{id} \times c_1) + (\mathit{id} \times c_2)) \fatsemi \mathit{factor}$$

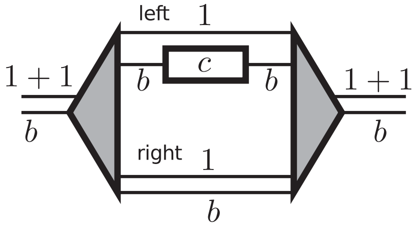

1
$c_1 : b_1 \leftrightarrow b_2$
bool
b_1
+
bool
b_2
1
$c_2 : b_1 \leftrightarrow b_2$

The diagram above shows the input value of type $(1+1) \times b$ processed by the distribute operator $\mathit{distrib}$, which converts it into a value of type $(1 \times b) + (1 \times b)$. In the left branch, which corresponds to the case when the boolean is true (i.e. the value was left ()), the combinator c_1 is applied to the value of type b. The right branch which corresponds to the boolean being false passes the value of type b through the combinator c_2.

Logic Gates. There are several universal primitives for conventional (irreversible) hardware circuits, such as nand and fanout. In the case of reversible hardware circuits, the canonical universal primitive is the Toffoli gate [Toffoli, 1980]. The Toffoli gate takes three boolean inputs: if the first two inputs are true then the third bit is negated. In a traditional language, the Toffoli gate would be most conveniently expressed as a conditional expression like:

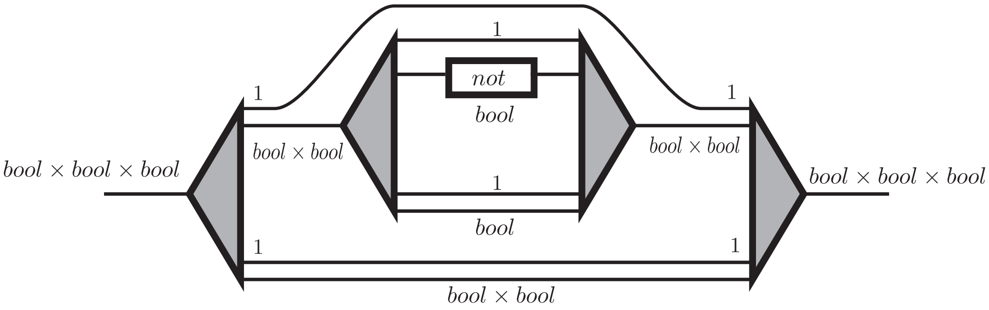
$\mathit{toffoli}(v_1, v_2, v_3) = \mathit{if}\ (v_1\ \mathit{and}\ v_2)\ \mathit{then}\ (v_1, v_2, \mathit{not}(v_3))\ \mathit{else}\ (v_1, v_2, v_3)$

We will derive Toffoli gate in Π by first deriving a simpler logic gate called cnot. Consider a one-armed version, if_c, of the conditional derived above. If the bool is true, the value of type b is modified by the operator c.



By choosing b to be bool and c to be not, we have the combinator $\mathit{if}_{\mathit{not}} : \mathit{bool} \times \mathit{bool} \leftrightarrow \mathit{bool} \times \mathit{bool}$ which negates its second argument if the first argument is true. This gate $\mathit{if}_{\mathit{not}}$ is often referred to as the cnot gate[Toffoli, 1980].

If we iterate this construction once more, the resulting combinator $\mathit{if}_{\mathit{cnot}}$ has type $\mathit{bool} \times (\mathit{bool} \times \mathit{bool}) \leftrightarrow \mathit{bool} \times (\mathit{bool} \times \mathit{bool})$. The resulting gate checks the first argument and if it is true, proceeds to check the second argument. If that is also true then it will negate the third argument. Thus $\mathit{if}_{\mathit{cnot}}$ is the required Toffoli gate.



Proposition 4.1. *Universality.* Π *is universal for combinational circuits.*

PROOF. This follows from the construction of the universal Toffoli gate above. □

Proposition 4.2. *Termination. Evaluation of well-typed combinators always terminates. In other words, for* $c : b_1 \leftrightarrow b_2$ *and* $v : b_1$ *there exists* $v' : b_2$ *such that* $c\ v \mapsto v'$.

PROOF. Termination follows directly from the fact that the operational semantics in definition 2.4 are defined inductively in terms of c – i.e. the size of the combinator strictly decreases. □

5. Categorical Structure

The language Π has an elegant semantics in *dagger symmetric monoidal categories* in two ways: with $+$ as the monoidal tensor or with $\times$ as the monoidal tensor. Also tensor $\times$ distributes over tensor $+$. Such a category is also called a *symmetric bimonoidal category*. We refer the reader to Chapter 2 for the required background concepts and definitions of category theory.

To be pedantic, we show that the *term model* of Π that follows from the extensional operational equality of combinators (see Def. 2.6) has a symmetric bimonoidal structure. The types of Π have the algebraic structure of a commutative semiring and hence the resulting categorical structure may be described as the *categorification of a commutative semiring*. This notion of "categorification" of algebraic structures has been popularized in recent years though the works of Baez and Dolan [1998], Crane and Yetter [1996] – the general idea is to find "category-theoretic analogs of set-theoretic concepts by replacing sets with categories, functions with functors, and equations between functions by natural isomorphisms between functors, which in turn should satisfy certain equations of their own, called coherence laws"[Baez and Dolan, 1998]. From this point of view, our work may be viewed as giving computational interpretations to simple algebraic structures and Π may be viewed as *computing in a commutative semiring*.

Finally a note on conventions: Since our primary interest is computation rather than category theory, whenever conventions between these fields have been in conflict, we have made notational choices in favor of computer science. For instance the action of doing f and then doing g is written as $f \fatsemi g$ whereas the convention in mathematics and category theory is to write $g \circ f$. Similarly, in our diagrams we choose to drawn the type $b_1 \times b_2$ with the b_1 wire on top and the b_2 wire below, whereas in categorical string diagrams sometimes the opposite convention is followed. Such choices can result in considerable comfort/discomfort based on the background of the reader and we apologize to those who may find uncomfortable. However these are entirely stylistic choices and do not affect the underlying theory in any way.

Proposition 5.1. *The language* Π *can be interpreted as a category whose objects are the base types,* b*, and whose morphisms are (equivalence classes of) combinators,* $c : b_1 \leftrightarrow b_2$*.*

Proof. For every morphism $c : b_1 \leftrightarrow b_2$ we have source and target objects as b_1 and b_2. Composition $g \circ f$ of morphisms $f : b_1 \leftrightarrow b_2$ and $g : b_2 \leftrightarrow b_3$ is given by sequencing $f \fatsemi g$. Further, we can check:

(1) Every object b has an identity morphism $id : b \leftrightarrow b$.

(2) Composition respects identity: $id \fatsemi f = f = f \fatsemi id$. Assuming $v_1 : b_1$, $v_2 : b_2$, $f\ v_1 \mapsto v_2$ we can check the required equality by writing out the derivation trees corresponding to evaluation of the combinators:

$$\frac{id\ v_1 \mapsto v_1 \quad f\ v_1 \mapsto v_2}{id \fatsemi f\ v_1 \mapsto v_2} \qquad\qquad \frac{f\ v_1 \mapsto v_2 \quad id\ v_2 \mapsto v_2}{id \fatsemi f\ v_1 \mapsto v_2}$$

(3) Associativity of composition follows from operational equivalence of $f \fatsemi (g \fatsemi h)$ and $(f \fatsemi g) \fatsemi h$ (where $h : b_3 \leftrightarrow b_4$). Assuming $v_3 : b_3$, $v_4 : b_4$, $g\ v_2 \mapsto v_3$ and $h\ v_3 \mapsto v_4$, one can check:

$$\frac{f\ v_1 \mapsto v_2 \qquad \dfrac{g\ v_2 \mapsto v_3 \quad h\ v_3 \mapsto v_4}{g \fatsemi h\ v_2 \mapsto v_4}}{f \fatsemi (g \fatsemi h)\ v_1 \mapsto v_4} \qquad\qquad \frac{\dfrac{f\ v_1 \mapsto v_2 \quad g\ v_2 \mapsto v_3}{f \fatsemi g\ v_1 \mapsto v_3} \qquad h\ v_3 \mapsto v_4}{(f \fatsemi g) \fatsemi h\ v_1 \mapsto v_4}$$

□

Proposition 5.2 (Dagger)**.** Π *is a dagger category.*

Proof. Every morphism $f : b_1 \leftrightarrow b_2$ has the adjoint $f^{\dagger} : b_2 \leftrightarrow b_1$. The following properties hold (where $g : b_2 \leftrightarrow b_3$):

(1) $id^{\dagger} = id : b \leftrightarrow b$.

(2) $(f \fatsemi g)^{\dagger} = g^{\dagger} \fatsemi f^{\dagger} : b_3 \leftrightarrow b_1$.

(3) $f^{\dagger\dagger} = f : b_1 \leftrightarrow b_2$.

The proof of $(f \fatsemi g)^{\dagger}$ and $f^{\dagger\dagger}$ equality follows from type directed induction and is covered in proof of logical reversibility (see Prop. 2.4). □

Proposition 5.3 (Symmetric Monoidal $(+, 0)$). Π *is a symmetric monoidal category with tensor* $+$ *and monoidal unit* 0*. The monoidal operation on morphisms is the additive composition of combinators* $c_1 + c_2$.

PROOF. We can check that $+$ is a tensor (usually denoted $\otimes$) with 0 as its unit (usually denoted I): for any two objects b_1 and b_2 we have the object $b_1 + b_2$ and for any two morphisms $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_3 \leftrightarrow b_4$, we have the morphism corresponding to the additive parallel composition $c_1 + c_2 : b_1 + b_3 \leftrightarrow b_2 + b_4$. To establish that category is monoidal one must show isomorphisms:

1. $b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3$ given by $assocl^{+}$.
2. $0 + b \leftrightarrow b$ given by $zeroe$. This is the equivalent of $\lambda_A : I \otimes A \rightarrow A$.
3. $b + 0 \leftrightarrow b$ given by $swap^{+} \fatsemi zeroe$. This is the equivalent of $\rho_A : A \otimes I \rightarrow A$.
4. We need to check that $+$ is a bifunctor.
   (a) $id_{b_1 \leftrightarrow b_1} + id_{b_2 \leftrightarrow b_2} = id_{b_1 + b_2 \leftrightarrow b_1 + b_2}$.
   (b) $(f + g) \fatsemi (j + k) = (f \fatsemi j) + (g \fatsemi k)$.
5. We need to check the naturality of $assocl^{+}$, $zeroe$ and $swap^{+} \fatsemi zeroe$.
   (a) $assocl^{+} \fatsemi ((f + g) + h) = (f + (g + h)) \fatsemi assocl^{+}$.
   (b) $zeroe \fatsemi f = (id + f) \fatsemi zeroe$.
   (c) $(swap^{+} \fatsemi zeroe) \fatsemi f = (f + id) \fatsemi (swap^{+} \fatsemi zeroe)$.
6. Satisfy certain coherence conditions which are usually called the "pentagon" and "triangle" axioms (see Sec 3.1 [Selinger, 2011])

The last three points require checking equality of combinators by writing out their derivation trees as we did in the case of associativity of sequential composition. To show symmetry, we need a braiding operation $b_1 + b_2 \leftrightarrow b_2 + b_1$ which is given by $swap^{+}$ (see Sec. 3.3 and 3.5 [Selinger, 2011]).

1. The braiding must satisfy two "hexagon" axioms.
2. The braiding is self inverse, $swap^{+}_{b_1 + b_2} = (swap^{+}_{b_2 + b_1})^{\dagger}$.

□

Proposition 5.4 (Symmetric Monoidal $(\times, 1)$)**.** Π *is a symmetric monoidal category with* $(\times, 1)$ *as the monoid.*

PROOF. This proof follows exactly the same pattern as with the $(+, 0)$ monoid. □

Proposition 5.5. Π *is a groupoid.*

PROOF. This follows by definition, since every morphism in Π is an isomorphism. □

Some technical and pedantic comments are due at this point.

- **Initial and Terminal objects.** The category Π has no initial and terminal objects. The objects 0 and 1 would be initial and terminal if we admitted all functions, such as in the category **Fin**.
- **Products and Coproducts.** The category Π has neither categorical products, nor categorical coproducts, i.e. $\times$ and $+$ are "merely" monoidal tensors. This follows from the fact that injection and projection maps are not isomorphisms. In other words, the category Π is neither cartesian nor cocartesian.
- **Trace.** In Chapter. 4 Sec. 1.1 we will see that a *trace* operator can be admitted in this category without any change of expressiveness.
- **String Diagrams.** In category theory, string diagrams have been developed as an alternate formal notation for categorical constructions. This is the same role served by Π's wiring diagrams. However due to their close similarity with string diagrams we don't rigorously formalize our graphical notation in this work.

  To establish wiring diagrams rigorously as a formal notation, we will need to show that equivalent diagrams for syntactically different combinators, such as $(f+g) \fatsemi (j+k)$ and $(f \fatsemi j)+(g \fatsemi k)$, do respect operational equivalence and that one may reason about such equivalences graphically. Also we will need to show that different layouts of the same combinator are equivalent. This involves reasoning about when it is valid to slide one wire over the other etc. Joyal et. al's work on "planar isotopy" [Joyal et al., 1996], Selinger [2011] and Mellies's "functorial boxes" [Melliès, 2006] show how this has been addressed before in the categorical setting.

While the graphical notation has been useful, in fact essential, in developing new constructions and reasoning about program evaluation, one might imagine that rigorous formalization is necessary to develop graphical IDE's and other mechanized tools for dealing with Π programs in the future.

4

Partiality and Turing Completeness

In this chapter we introduce the information preserving logically reversible computational model called Π^o. The main difference between Π, which was complete for (finite) combinational circuits is that Π^o is Turing complete. Computations definable in Π^o are partial isomorphisms. This means two specific things (1) Π^o admits non-terminating computation and (2) all terminating computations are however logically reversible. Further Π^o computations can relate types that do not have the same number of inhabitants and hence must be thought of as defining isomorphisms between the strict domain and strict co-domain of the computation type.

Thus Π^o has a very different flavor from the strongly normalizing language Π. To indicate that computation in Π^o may be partial in both directions, Π^o types have the form $b_1 \rightleftharpoons b_2$, as opposed to Π types of the form $b_1 \leftrightarrow b_2$. All the $b_1 \leftrightarrow b_2$ combinators we have constructed so far however may be thought of as $b_1 \rightleftharpoons b_2$ combinators since the later is strictly a superset.

We derive Π^o from Π by extending it with *trace* operators from category theory and isorecursive types. Adding *trace* operators allows us to express iteration/feedback loops in Π^o thereby allowing computations to be executed repeatedly. Adding recursive types allows us to express types with infinite inhabitants like numbers and lists. As before, we examine constructions in the new system and state some of its properties. These extensions are also carried over into the graphical language where, as before, computation is the movement of particles denoting values in a circuit denoting the combinator acting on the value.

1. Traces and Isorecursive types

From the categorical notion of traces [Joyal et al., 1996], we extract a *trace* operator that acts over the monoid $(+, 0)$ with the type judgment and big-step operational semantics given below.

Definition 1.1 (Type Judgment for *trace*).

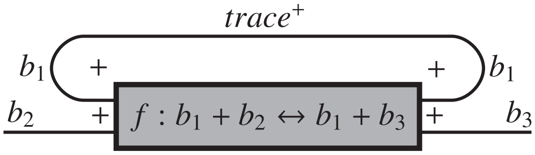
$$\frac{c : b_1 + b_2 \rightleftharpoons b_1 + b_3}{\mathit{trace}\ c : b_2 \rightleftharpoons b_3}$$

Intuitively, given a computation $c : b_1 + b_2 \rightleftharpoons b_1 + b_3$ we *cancel* the common type b_1 from both the input and the output. The resulting combinator is $\mathit{trace}\ c : b_2 \rightleftharpoons b_3$. Consequently, trace is also sometimes called a *cancellation law*. Formally, we express its semantics as a loop.

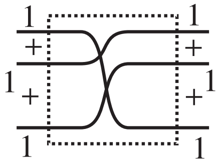
Definition 1.2 (Operational Semantics for trace)**.**

$$\frac{(c \fatsemi \mathit{loop}_c)\ (\mathit{right}\ v_1) \mapsto v_2}{(\mathit{trace}\ c)\ v_1 \mapsto v_2} \qquad \frac{(c \fatsemi \mathit{loop}_c)\ (\mathit{left}\ v_1) \mapsto v_2}{\mathit{loop}_c\ (\mathit{left}\ v_1) \mapsto v_2} \qquad \frac{}{\mathit{loop}_c\ (\mathit{right}\ v) \mapsto v}$$

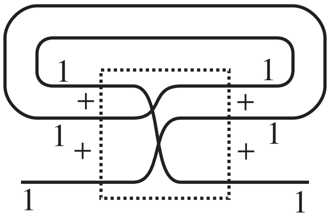
In the above rules loop_c denotes an internal cyclic version of a given combinator c. It is 'internal' in the sense that loop_c is not available in the syntax of Π^o as a combinator that the user has access to and should be thought of as a syntactic convenience for describing the semantics. As the semantics show, the value of type b_2 is injected into the sum type $b_1 + b_2$ by tagging it with the right constructor. The tagged value is passed to c. As long as c returns a value that is tagged with left, that tagged value is fed back to c. As soon as a value tagged with right is returned, that value is returned as the final answer of the $\mathit{trace}\ c$ computation. The wiring diagram for trace has a loop (annotated with type b_1) showing the iteration.



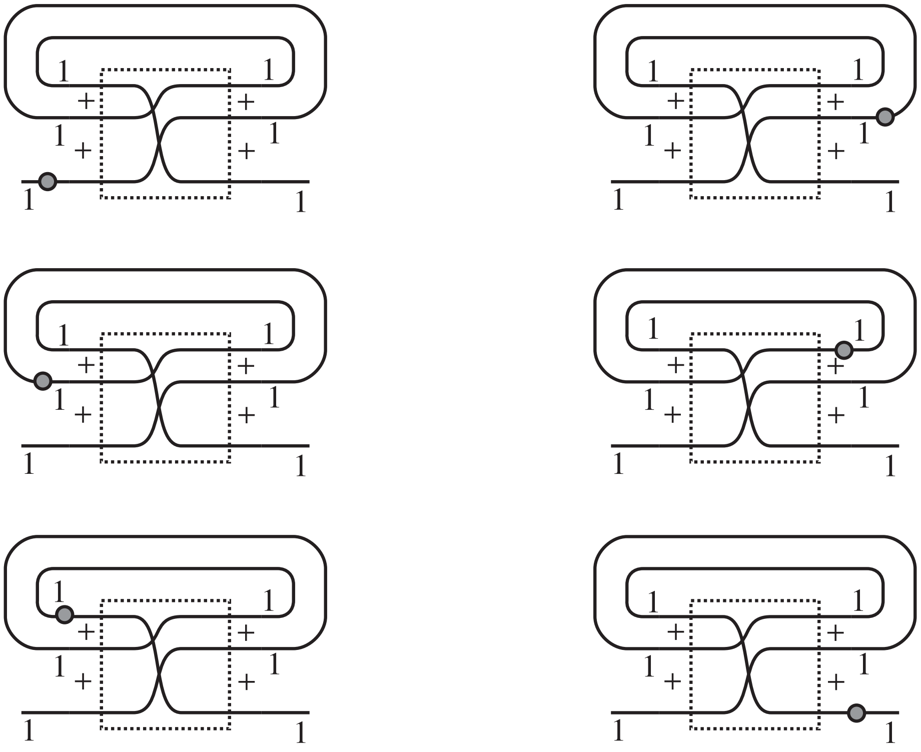
Example. Let us examine the execution of a simple circuit $c\ :\ \mathit{bool} + 1\ \rightleftharpoons\ \mathit{bool} + 1$ when traced. While we could do this syntactically, it is more intuitive to use the graphical language.



When we trace c we have the circuit $\mathit{trace}\ c : 1 \rightleftharpoons 1$ shown below.



The circuit may be executed by examining the flow of a value of type 1 through it. The following diagrams are the state of the execution of the circuit at different points in time, showing where the value is at each point.



The value starts at the input to the circuit, flows out on the *left* branch of c and is traced back into c. The feedback loop is taken exactly two times in the execution of this circuit.

Adding *trace* to Π does not make the system Turing complete or introduce infinite loops. Since the types are finite there is only a finite number of states the program can be in.

The definition of traced categories is expressed using a particular monoidal tensor. We have two possible tensors in Π which in principle can be extended to support trace operators. The natural tensor to use in this case, however, is the sum operator $+$ as it leads to usual looping constructs as explained next. Traces over the monoid $(\times, 1)$ have been studied as a basis for fixpoints [Hasegawa, 1997] and extending Π with those would a topic for future work (see Chapter 8).

1.1. Traces in Π**.** The language of isomorphisms, Π^o, already contained an implicit *trace* operator. This follows from the fact that Π is complete for isomorphisms of finite types. For any given combinator $c : b_1 + b_2 \leftrightarrow b_1 + b_3$ in Π, we can always construct the (terminating) combinator *trace* $c : b_2 \leftrightarrow b_3$ with the desired semantics. Thus adding an explicit *trace* operation to Π does not change its expressive power. Consequently the following proposition holds:

Proposition 1.3. Π *has the structure required for a traced dagger symmetric monoidal categories.*

1.2. Isorecursive Types. To express computations over natural numbers and other types that have non-finite inhabitants, we extend the Π types with isorecursive type definitions of the form $\mu x.b$. Here 'μ' binds the recursive type variable 'x' that may occur free in the type 'b'.

Isorecursive types fit naturally within the framework of a reversible language as they come equipped with two isomorphisms *fold* and *unfold* that witness the equivalence of a value of a recursive type with its "unrollings".

Definition 1.4 (Type judgments for *fold/unfold*)**.**

$$\mathit{fold} : \quad b[\mu x.b/x] \rightleftharpoons \mu x.b \quad : \mathit{unfold}$$

The notation $b[\mu x.b/x]$ denotes the capture free substitution of x with $\mu x.b$ in the expression b, similar to the substitution operation used in the definition of β-reduction. Similarly, to construct a value of type $\mu x.b$, we must first have a value of type $b[\mu x.b/x]$ and we use *fold* to construct the packed form of the value.

To create recursive values, we introduce the notation $\langle v \rangle$ for values, with the following type rule:

Definition 1.5 (Type judgments for packed values)**.**

$$\frac{\vdash v : b[\mu x.b/x]}{\vdash \langle v \rangle : \mu x.b}$$

A value of the form $\langle v \rangle$, denotes a packed recursive value that has the type $\mu x.b$. To do any transformation on such a value one must first unpack the value using $unfold$. Thus we have the reduction relations:

Definition 1.6 (Operational semantics for isorecursive types)**.**

$$\begin{array}{rcll} \mathit{fold} & v & \mapsto & \langle v \rangle \\ \mathit{unfold} & \langle v \rangle & \mapsto & v \end{array}$$

1.3. Example: Numbers. As an example consider the inductive definition of the type of natural which we would define in a language like Haskell as:

```
data Nat = Zero | Succ Nat
```

The above Haskell type is the equivalent of the Π^o type $\mu x.(1 + x)$, which we will abbreviate as nat. The type $\mu x.1 + x$ represents the natural numbers in unary format. For nat, the unfolded type $b[\mu x.b/x]$ is $(1+x)[\mu x.(1+x)/x]$. After substitution this is $1+\mu x.(1+x)$ or equivalently $1 + nat$.

The *fold* rule says that a value of the type $b[\mu x.b/x]$ can be converted to a value of the type $\mu x.b$. In the case of nat, this means that $1 + nat$ can be converted to nat by *fold*. The simplest value of the type $1 + nat$ is *left* () and hence we have the number $0 : nat$ represented by $\langle \mathit{left}\ () \rangle : nat$ which is the result of $\mathit{fold}\ (\mathit{left}\ ())$.

Consider the table of values below. The operation *fold* maps values on the left (all of which have the type $1 + nat$) to values on the middle column (all of which have the type nat). The third column is our numeric interpretation of the values in the middle column. Similarly, $unfold$ at the type nat would map values in the middle column to values in the left column.

$\mathit{left}\ ()$	$\langle \mathit{left}\ () \rangle$	0
$\mathit{right}\ \langle \mathit{left}\ () \rangle$	$\langle \mathit{right}\ \langle \mathit{left}\ () \rangle \rangle$	1
$\mathit{right}\ \langle \mathit{right}\ \langle \mathit{left}\ () \rangle \rangle$	$\langle \mathit{right}\ \langle \mathit{right}\ \langle \mathit{left}\ () \rangle \rangle \rangle$	2
...	...	...

When we unfold a number $n : \mathit{nat}$ to a value of $1 + \mathit{nat}$, the value either has the form $\mathit{left}\ ()$ or $\mathit{right}\ n'$. If the value has the type $\mathit{left}\ ()$, then we know that n was 0. If however the value we get has the form $\mathit{right}\ n'$ then numerically $n' = n - 1$.

Just like with λ-calculus, once we show that numbers can be encoded in the system, it is handy to think of numbers as being a built-in. Hence in later discourse we will just use the usual numerals 0,1,2.. for numbers and not bother with their specific encoding.

1.4. Example: Lists. Consider the Haskell definition of a list of numbers shown below.

```
data L = Nil | Cons Nat L
```

In Π^o we define this type as $\mu x.(1 + \mathit{nat} \times x)$. We use the name list as an abbreviation for $\mu x.1 + \mathit{nat} \times x$, As before, we have $\mathit{fold} : 1 + \mathit{nat} \times \mathit{list} \rightleftharpoons \mathit{list} : \mathit{unfold}$.

The table below has three columns. The leftmost shows values of the type $b[\mu x.b/x]$, the column in the middle shows values of the $\mu x.b$ and the column on the right is our intuitive reading of the value.

$\mathit{left}\ ()$	$\langle \mathit{left}\ () \rangle$	$[]$
$\mathit{right}\ (5, \langle \mathit{left}\ () \rangle)$	$\langle \mathit{right}\ (5, \langle \mathit{left}\ () \rangle) \rangle$	$[5]$
$\mathit{right}\ (3, \langle \mathit{right}\ (5, \langle \mathit{left}\ () \rangle) \rangle)$	$\langle \mathit{right}\ (3, \langle \mathit{right}\ (5, \langle \mathit{left}\ () \rangle) \rangle) \rangle$	$[3, 5]$
...	...	...

Isorecursive types as explained here are different from *equirecursive* types studied in the literature. The important difference is that, in the case of equirecursive types, there is no explicit isomorphism that mediates between $1 + \mathit{nat}$ and nat (like fold and unfold do here). Thus the value $\mathit{left}\ ()$ has both the type $1 + \mathit{nat}$ and nat, and there is no explicit packed form $\langle \mathit{left}\ () \rangle$. See Chapter 20 of Pierce [2002] for a detailed discussion.

2. The Computational Model of Partial Isomorphisms : Π^o

We have extend Π in two dimensions: (i) by adding recursive types and (ii) by adding looping constructs. The combination of the two extensions makes the extended language, Π^o, expressive enough to write arbitrary looping programs, including non-terminating ones. We collect together the extensions to provide the full syntactic and semantic definition:

Definition 2.1 (Syntax of Π^o)**.** *We collect our types, values, and combinators, to get the full language definition.*

$$\begin{array}{rcl}
\mathit{value\ types}, b & ::= & 0 \mid 1 \mid b + b \mid b \times b \mid \mu x.b \mid x \\
\mathit{values}, v & ::= & () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \mid \langle v \rangle \\
\\
\mathit{combinator\ types}, t & ::= & b \rightleftharpoons b \\
\mathit{isomorphisms}, \mathit{iso} & ::= & \mathit{zeroe} \mid \mathit{zeroi} \mid \mathit{swap}^{+} \mid \mathit{assocl}^{+} \mid \mathit{assocr}^{+} \\
& \mid & \mathit{unite} \mid \mathit{uniti} \mid \mathit{swap}^{\times} \mid \mathit{assocl}^{\times} \mid \mathit{assocr}^{\times} \\
& \mid & \mathit{distrib}_0 \mid \mathit{factor}_0 \mid \mathit{distrib} \mid \mathit{factor} \mid \mathit{fold} \mid \mathit{unfold} \\
\mathit{combinators}, c & ::= & \mathit{iso} \mid \mathit{id} \mid \mathit{sym}\ c \mid c \fatsemi c \mid c + c \mid c \times c \mid \mathit{trace}\ c
\end{array}$$

Definition 2.2 (Type judgments for values)**.**

$$\frac{}{\vdash () : 1} \qquad \frac{\vdash v_1 : b_1 \quad \vdash v_2 : b_2}{\vdash (v_1, v_2) : b_1 \times b_2}$$

$$\frac{\vdash v : b_1}{\vdash \mathit{left}\ v : b_1 + b_2} \qquad \frac{\vdash v : b_2}{\vdash \mathit{right}\ v : b_1 + b_2} \qquad \frac{\vdash v : b[\mu x.b/x]}{\vdash \langle v \rangle : \mu x.b}$$

Definition 2.3 (Type judgment for combinators)**.**

$$
\begin{array}{rccl}
\mathit{zeroe} : & 0 + b \leftrightarrow b & : \mathit{zeroi} \\
\mathit{swap}^{+} : & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \mathit{swap}^{+} \\
\mathit{assocl}^{+} : & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : \mathit{assocr}^{+} \\
\mathit{unite} : & 1 \times b \leftrightarrow b & : \mathit{uniti} \\
\mathit{swap}^{\times} : & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : \mathit{swap}^{\times} \\
\mathit{assocl}^{\times} : & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & : \mathit{assocr}^{\times} \\
\mathit{distrib}_0 : & 0 \times b \leftrightarrow 0 & : \mathit{factor}_0 \\
\mathit{distrib} : & (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) & : \mathit{factor} \\
\mathit{fold} : & b[\mu x.b/x] \rightleftharpoons \mu x.b & : \mathit{unfold}
\end{array}
$$

$$
\frac{}{\mathit{id} : b \leftrightarrow b}
\qquad
\frac{c : b_1 \leftrightarrow b_2}{\mathit{sym}\ c : b_2 \leftrightarrow b_1}
\qquad
\frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3}
$$

$$
\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4}
\quad
\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4}
\quad
\frac{c : b_1 + b_2 \rightleftharpoons b_1 + b_3}{\mathit{trace}\ c : b_2 \rightleftharpoons b_3}
$$

Definition 2.4 (Operational semantics for Π^o)**.** *Given a program $c : b_1 \rightleftharpoons b_2$ in Π^o, we can run it by supplying it with a value $v_1 : b_1$. The evaluation rules $c\ v_1\ \mapsto v_2$ are given below.*

Additive fragment:

$$
\begin{array}{rlcl}
\mathit{zeroe} & (\mathit{right}\ v) & \mapsto & v \\
\mathit{zeroi} & v & \mapsto & \mathit{right}\ v \\
\mathit{swap}^{+} & (\mathit{left}\ v) & \mapsto & \mathit{right}\ v \\
\mathit{swap}^{+} & (\mathit{right}\ v) & \mapsto & \mathit{left}\ v \\
\mathit{assocl}^{+} & (\mathit{left}\ v_1) & \mapsto & \mathit{left}\ (\mathit{left}\ v_1) \\
\mathit{assocl}^{+} & (\mathit{right}\ (\mathit{left}\ v_2)) & \mapsto & \mathit{left}\ (\mathit{right}\ v_2) \\
\mathit{assocl}^{+} & (\mathit{right}\ (\mathit{right}\ v_3)) & \mapsto & \mathit{right}\ v_3 \\
\mathit{assocr}^{+} & (\mathit{left}\ (\mathit{left}\ v_1)) & \mapsto & \mathit{left}\ v_1 \\
\mathit{assocr}^{+} & (\mathit{left}\ (\mathit{right}\ v_2)) & \mapsto & \mathit{right}\ (\mathit{left}\ v_2) \\
\mathit{assocr}^{+} & (\mathit{right}\ v_3) & \mapsto & \mathit{right}\ (\mathit{right}\ v_3)
\end{array}
$$

Multiplicative fragment:

$$\begin{array}{rlcl}
\mathit{unite} & ((),v) & \mapsto & v \\
\mathit{uniti} & v & \mapsto & ((),v) \\
\mathit{swap}^{\times} & (v_1,v_2) & \mapsto & (v_2,v_1) \\
\mathit{assocl}^{\times} & (v_1,(v_2,v_3)) & \mapsto & ((v_1,v_2),v_3) \\
\mathit{assocr}^{\times} & ((v_1,v_2),v_3) & \mapsto & (v_1,(v_2,v_3))
\end{array}$$

Distributivity and factoring:

$$\begin{array}{rlcl}
\mathit{distrib} & (\mathit{left}\ v_1,v_3) & \mapsto & \mathit{left}\ (v_1,v_3) \\
\mathit{distrib} & (\mathit{right}\ v_2,v_3) & \mapsto & \mathit{right}\ (v_2,v_3) \\
\mathit{factor} & (\mathit{left}\ (v_1,v_3)) & \mapsto & (\mathit{left}\ v_1,v_3) \\
\mathit{factor} & (\mathit{right}\ (v_2,v_3)) & \mapsto & (\mathit{right}\ v_2,v_3)
\end{array}$$

Folding and Unfolding:

$$\begin{array}{rccc}
\mathit{fold} & v & \mapsto & \langle v\rangle \\
\mathit{unfold} & \langle v\rangle & \mapsto & v
\end{array}$$

The evaluation rules of the composition combinators are given below:

$$\frac{}{\mathit{id}\ v\mapsto v}\quad \frac{c^{\dagger}v_1\mapsto v_2}{(\mathit{sym}\ c)\ v_1\mapsto v_2}\quad \frac{c_1\ v_1\mapsto v\quad c_2\ v\mapsto v_2}{(c_1\mathbin{;}c_2)\ v_1\mapsto v_2}$$

$$\frac{c_1\ v_1\mapsto v_2}{(c_1+c_2)\ (\mathit{left}\ v_1)\mapsto \mathit{left}\ v_2}\quad \frac{c_2\ v_1\mapsto v_2}{(c_1+c_2)\ (\mathit{right}\ v_1)\mapsto \mathit{right}\ v_2}$$

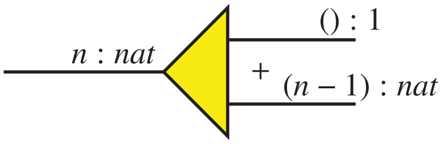
$$\frac{c_1\ v_1\mapsto v_3\quad c_2\ v_2\mapsto v_4}{(c_1\times c_2)\ (v_1,v_2)\mapsto (v_3,v_4)}$$

$$\frac{(c\mathbin{;}\mathit{loop}_c)\ (\mathit{right}\ v_1)\mapsto v_2}{(\mathit{trace}\ c)\ v_1\mapsto v_2}\quad \frac{(c\mathbin{;}\mathit{loop}_c)\ (\mathit{left}\ v_1)\mapsto v_2}{\mathit{loop}_c\ (\mathit{left}\ v_1)\mapsto v_2}\quad \frac{}{\mathit{loop}_c\ (\mathit{right}\ v)\mapsto v}$$

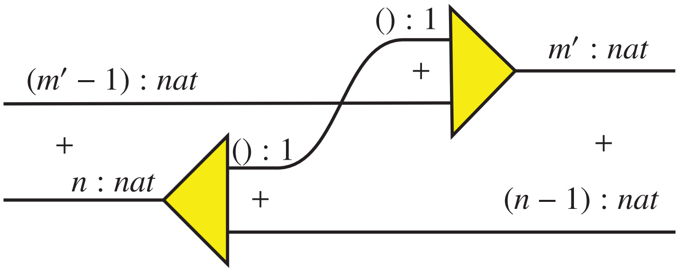
3. Constructions in Π^o

We now present several programming examples that illustrate the expressiveness of Π^o. These examples establish idiomatic constructions that will be used in subsequent chapters. In the code below we use bool as abbreviation for $1+1$ and nat as abbreviation for $\mu x.1+x$. In most cases, we present the constructions using diagrams.

3.1. Addition and Subtraction. The fold isomorphism at the type nat can be represented graphically as shown below. The unfold is its corresponding mirror image.



The first interesting combinator to consider is one below that does addition *or* subtraction, $\mathit{add} : \mathit{nat} + \mathit{nat} \rightleftharpoons \mathit{nat} + \mathit{nat}$. For $v : \mathit{nat}$, if the combinator is applied to $\mathit{left}\ v$ the output is essentially $\mathit{left}\ (v+1)$. If the v is non-zero and we apply the combinator to $\mathit{right}\ v$, we get the output $\mathit{right}\ (v-1)$. If v is 0, then we have $\mathit{left}\ 0$.



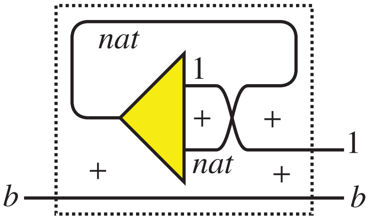
Based on the above we can write out a map for the values:

$$\left\| \begin{array}{c|c} \mathit{left}\ n & \mathit{left}\ n+1 \\ \mathit{right}\ 0 & \mathit{left}\ 0 \\ \mathit{right}\ n+1 & \mathit{right}\ n \end{array} \right\|$$

While is clear that the above combinator is clearly an isomorphism, it also accomplishes addition and subtraction as long the operations are well defined. This combinator however is interesting because it serves as the basis for the construction of a total function add_1 and its adjoint sub_1 which is a partial function that is undefined on input 0.

3.2. Partiality and Non-termination. We use the basic idea of the addition or subtraction construction above to construct the just combinator below. The combinator $\mathit{just} : b \rightleftharpoons 1+b$ injects a value into a larger type. In a sense this combinator may be viewed as a strange sort of *identity* function.

The combinator takes an input of an arbitrary type b and passes it essentially unchanged to the output. The output type is larger because we used trace to hide the unfolding of an implicit nat. Since the input value of type b inhabits the $right$ branch at runtime, no such nat really exists.



This combinator is significant for two reasons:

(1) The type b is not isomorphic to $b + 1$. Thus Π^o combinators can relate non-isomorphic types. This was not true of Π combinators.

(2) The size of $b + 1$ is one more than the size of b. Hence there is at least one value for which the adjoint of $just$ is undefined. It is easy to see that $just$ maps every $v : b$ to $right\ v : 1 + b$. Thus, $just^\dagger : 1 + b \rightleftharpoons b$ diverges on $left\ ()$ showing that Π^o admits non-terminating computations. This was again not true of Π combinators.

The construction of $just$ reveals the nature of $trace$ as a unconstrained cancellation law of the form $(b_1 + b_2 = b_1 + b_3) \rightarrow (b_2 = b_3)$. In normal arithmetic, such a law is consistent only if the canceled quantity b_1 is finite. The size of nat is the size of the set of natural numbers. Hence non-termination and non-isomorphic computation is introduced by effectively canceling out an infinite type using the $trace$ operation.

3.3. Numeric Operations. Using $just$, we can conveniently write add_1 and sub_1 combinators of type $nat \rightleftharpoons nat$ as follows:

$$add_1 = just \mathbin{⨾} fold$$

$$sub_1 = sym\ add_1.$$

The definition implies that $sub_1\ 0$ diverges. Other arithmetic operations can be derived by using these combinators in conjunction with the bounded iteration construction.

3.4. Creating Constants. We can also create, for any particular value v, a constant function returning v. For example, we can trivially write functions that introduce the values $false$ and $true$ as:

$$introF, introT : 1 \rightleftharpoons bool$$

$intro F = just$

$intro T = just \fatsemi not$

This construction can be extended to create constants of any finite type. Given combinators $intro T$ and $intro F$, we can construct a combinator that injects a value into a left or right summand. For example,

$injectRight : a \rightleftharpoons a + a$

$injectRight = uniti \fatsemi (intro F \times id) \fatsemi distrib \fatsemi (unite + unite)$

We can introduce $0 : nat$ using $injectRight$ as follows:

$introZ : 1 \rightleftharpoons nat$

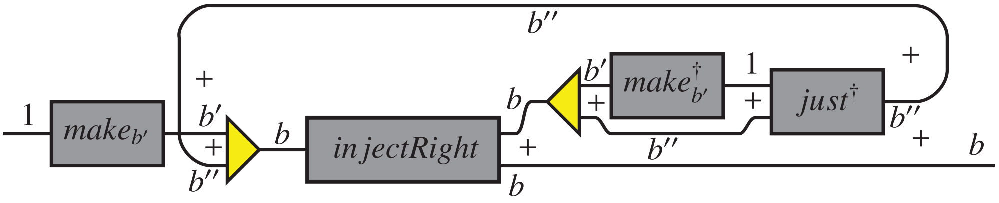
$introZ = trace\ (swap^{+} \fatsemi fold \fatsemi injectRight)$

Further we can create any $n : nat$ by $introZ \fatsemi add_1^n : 1 \rightleftharpoons nat$.

Most recursive types have the form $\mu x.(polynomial\ in\ x)$ and can be written as $\mu x.(b' + b'')$ where b' is the base case of the recursive type, that does not refer to x, and b'' is the component that refers to x. For example, for the type $nat = \mu x.1 + x$ we have $b' = 1$ and $b'' = x$; for $list = \mu x.1 + nat \times x$ we have $b' = 1$ and $b'' = nat \times x$.

We can generalize the construction of $0 : nat$ to the creation of constants of any recursive type b that has the form $\mu x.b' + b''$ as described above. To construct the combinator $make_b : 1 \rightleftharpoons b$ we assume have a combinator that makes a constant of the strictly smaller b', $make_b' : 1 \rightleftharpoons b'$.

The desired $make_b$ can be constructed as shown below. The combinator does the following: from the input () it creates $v : b''$ which is then folded into the desired $\langle v \rangle : b$. The combinator $injectRight$ subsequently injects it into $right\ \langle v \rangle$ which traced away at the output to get $\langle v \rangle : b$. The $trace$ feeds back the dangling b'' component which was assumed by $fold$.



It is important to note that $make_b : 1 \rightleftharpoons b$ constructs a specific (compile time) constant of type b. We can construct any specific constant our choice by composing several such $make$ combinators. The combinator $make$ does not generate a random inhabitant of the type b. Thus its adjoint, $make_b^{\dagger}$, erases a specific constant of type b. If the adjoint is supplied with any other value of type b it will diverge.

3.5. Logical Reversibility and Turing Completeness. As before, each combinator has an adjoint and the language is reversible.

Proposition 3.1 (Logical Reversibility). $c\ v \mapsto v'$ *iff* $c^{\dagger} v' \mapsto v$.

PROOF. The operators $fold$ and $unfold$ are adjoint to each other. The adjoint of $trace\ c$ is $trace\ c^{\dagger}$.

The only complexity in proof comes from reasoning about the reversibility of $trace$. We are interested in the reversibility of terminating execution sequences. Any terminating sequence of $trace\ c$ will have the following form, once the $loop_c$ applications have been expanded out:

$$c \fatsemi c \fatsemi c \fatsemi c \fatsemi c \fatsemi c ... c\ (right\ v') \mapsto (right\ v)$$

This is a finite sequence of c applications such that the result is of the form $right\ v$. The reversibility of this follows directly from the reversibility of c and sequencing. □

Proposition 3.2 (Turing Completeness). Π^o *is Turing complete.*

This proposition will be proved in subsequent chapters (See Chapter 7) where we take a high level Turing complete language and compile it to Π^o.

4. Categorical Structure

The language Π^o has an elegant semantics in *traced dagger symmetric monoidal categories* with '$+$' as the monoidal tensor.

Proposition 4.1. Π^o *is a symmetric monoidal category with* $(+, 0)$ *as the monoid.*

PROOF. The proof from Chapter 3 Prop. 5.3 essentially unchanged since the addition isorecursive types and the $trace$ operator do not change the underlying tensor. □

Proposition 4.2. Π^o *is a symmetric monoidal category with* $(\times, 1)$ *as the monoid.*

PROOF. This is similar to the $(+, 0)$ situation. □

Proposition 4.3. Π^o *is a dagger category.*

PROOF. This follows from the properties of the adjoint as with Chapter 3 Prop. 5.2. □

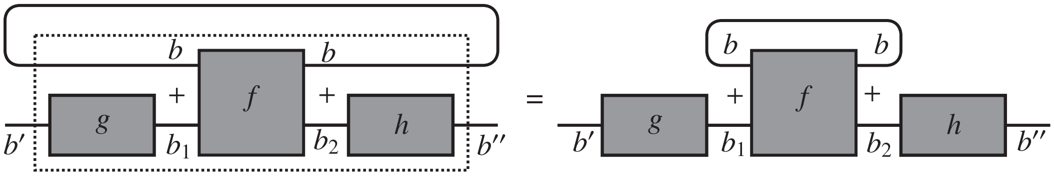
As before, despite having a monoidal structure with $(\times, 1)$ and with $(+, 0)$, Π^o is not a category with categorical products or sums because the required projection and injection functions do not exist. Neither does Π^o have initial or terminal objects because there is no object to/from which all other objects have morphisms.

Proposition 4.4. Π^o *is a traced dagger symmetric monoidal category with* $(+, 0)$ *as the monoid.*

PROOF. We need to show that the following *coherence conditions* hold. In each case we have drawn out the constructions and the required equality between them. It must be noted that the $trace$ used here is a *left trace* in category theory. However as pointed out in Sec. 5 Chapter 3, our drawing convention is the top-to-bottom mirror image of the equivalent categorical string diagrams.

The proof for most of the cases following directly from the operational equivalence of the combinators, which are easily shown by writing out the execution traces as before. We work out the details of the *tightening* case in detail. Others follow similarly.

(1) Tightening



Given $g : b' \rightleftharpoons b_1$, $f : b + b_1 \rightleftharpoons b + b_2$ and $h : b_2 \rightleftharpoons b''$, we need to show the equality

$$trace\ ((id + g) \fatsemi f \fatsemi (id + h)) = g \fatsemi trace\ f \fatsemi h.$$

at the type $b' \rightleftharpoons b''$.

This proof is sketched out as an informal induction to show the required bisimulation.

To show LHS = RHS, we must show that for all values $v' : b'$, either both LHS and RHS diverge or $LHS\ v' \mapsto v''$ and $RHSv' \mapsto v''$ where $v'' : b''$.

Termination. Let us focus on the terminating sequences for which we assume that every invocation of f, g and h terminate. Let $c = (id \times g) \fatsemi f \fatsemi (id \times h)$ such that we have:

$$\dfrac{\dfrac{\begin{array}{c} g\ v' \mapsto v'_1 \\ \vdots \end{array} \qquad \dfrac{\begin{array}{c} A \\ \vdots \end{array}}{(f \fatsemi (id + h) \fatsemi loop_c)\ v_1 \mapsto v''}\ (where\ v_1 = right\ v'_1)}{((id + g) \fatsemi f \fatsemi (id + h) \fatsemi loop_c)\ right\ v' \mapsto v''}}{(trace\ c)\ v' \mapsto v''}$$

The derivation of branch A depends on the execution of f. To reason about the iteration cased by trace, we have to reason about the intermediate values acted upon by f, the first of which is $right\ v'_1$. We denote such intermediate values by v_i where each v_i is a value of the form $left\ v'_i$ or $right\ v'_i$. The terminating value of the iteration sequence is denoted $right\ v'_k$.

If $f\ v_i$ produces a value of the form $left\ v'_{(n+1)}$, then:

$$\dfrac{f\ v_i \mapsto \mathit{left}\ v'_{i+1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{A}{(f \fatsemi (\mathit{id}+h) \fatsemi \mathit{loop}_c)\ v_{i+1} \mapsto v''}\ (\mathit{where}\ v_{i+1} = \mathit{left}\ v'_{i+1})}{(c \fatsemi \mathit{loop}_c)\ \mathit{left}\ v'_{i+1}}}{\mathit{loop}_c\ \mathit{left}\ v'_{i+1} \mapsto v''}}{((\mathit{id}+h) \fatsemi \mathit{loop}_c)\ \mathit{left}\ v'_{i+1} \mapsto v''}}{(f \fatsemi (\mathit{id}+h) \fatsemi \mathit{loop}_c)\ v_i \mapsto v''}$$

If $f\ v_i$ produces a value of the form $\mathit{right}\ v'_k$:

$$\dfrac{f\ v_i \mapsto \mathit{right}\ v'_k \qquad \dfrac{\begin{matrix} h\ v'_k \mapsto v'' \\ \vdots \end{matrix} \qquad \mathit{loop}_c\ (\mathit{right}\ v'') \mapsto v''}{((\mathit{id}+h) \fatsemi \mathit{loop}_c)\ \mathit{right}\ v'_{i+1} \mapsto v''}}{(f \fatsemi (\mathit{id}+h) \fatsemi \mathit{loop}_c)\ v_i \mapsto v''}$$

This completes the evaluation of v' to v'' by the LHS. To prove the desired equality we must show that $(g \fatsemi \mathit{trace}\ f \fatsemi h)\ v' \mapsto v''$ using the assumptions $g\ v' \mapsto v'_1$, $h\ v'_k \mapsto v''$ and for each iteration $f\ v_i \mapsto v_{i+1}$.

$$\dfrac{\dfrac{g\ v' \mapsto v'_1 \qquad \dfrac{\begin{matrix} A \\ \vdots \\ (f \fatsemi \mathit{loop}_f)\ v_1 \mapsto v'_k \end{matrix}}{(\mathit{trace}\ f)\ v'_1 \mapsto v'_k} \qquad h\ v'_k \mapsto v''}{(\mathit{trace}\ f \fatsemi h)\ v'_1 \mapsto v''}}{(g \fatsemi \mathit{trace}\ f \fatsemi h)\ v' \mapsto v''}$$

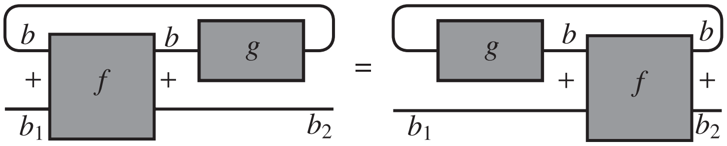
The derivation A depends on the behaviour of f. If $f\ v_i$ results in a value of the form $\mathit{left}\ v'_{i+1}$ then we have:

$$\dfrac{f\ v_i \mapsto \mathit{left}\ v'_{i+1} \qquad \dfrac{\begin{matrix} A \\ \vdots \\ (f \fatsemi \mathit{loop}_f)\ v_{i+1} \mapsto v'_k \end{matrix}}{\mathit{loop}_f\ \mathit{left}\ v'_{i+1} \mapsto v'_k}\ (\mathit{where}\ v_{i+1} = \mathit{left}\ v'_{i+1})}{(f \fatsemi \mathit{loop}_c)\ v_i \mapsto v'_k}$$

If $f\ v_i$ results in a value of the form $\mathit{right}\ v'_k$ then we have:

$$\dfrac{f\ v_i \mapsto \mathit{right}\ v'_k \qquad \mathit{loop}_f\ (\mathit{right}\ v'_k) \mapsto v'_k}{(f \fatsemi \mathit{loop}_c)\ v_i \mapsto v'_k}$$

(2) Sliding

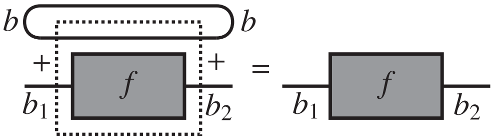


Given $f : b + b_1 \rightleftharpoons b + b_2$ and $g : b \rightleftharpoons b$, we have to show

$$trace\ (f \fatsemi (g + id)) = trace\ ((g + id) \fatsemi f)$$

at the type $b_1 \rightleftharpoons b_2$.

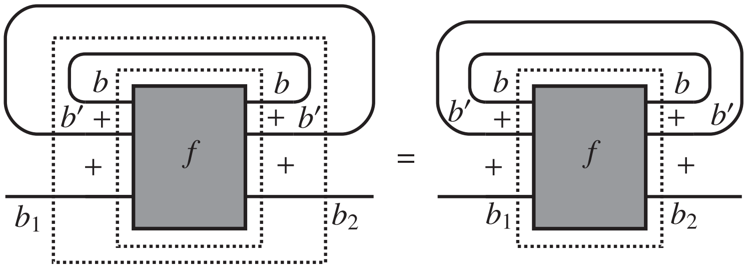
(3) Vanishing



Given $f : b_1 \rightleftharpoons b_2$ we have to show

$$trace\ (id + f) = f$$

at the type $b_1 \rightleftharpoons b_2$.

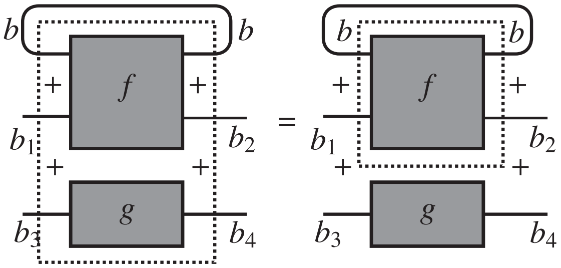


Given $f : b + (b' + b_1) \rightleftharpoons b + (b' + b_2)$ we have to show

$$trace\ (trace\ f) = trace\ (assocr^{+} \fatsemi f \fatsemi assocl^{+})$$

at the type $b_1 \rightleftharpoons b_2$.

(4) Strength

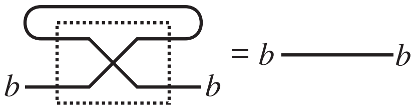


Given $f : b + b_1 \rightleftharpoons b + b_2$ and $g : b_3 \rightleftharpoons b_4$ we have to show that

$$trace\ (assocl^{+} \fatsemi (f + g) \fatsemi assocr^{+}) = (trace\ f) + g$$

at the type $b_1 + b_3 \rightleftharpoons b_2 + b_4$.

(5) Yanking



Given $swap^+ : b + b \rightleftharpoons b + b$ we have to show that:

$$trace\ swap^+ = id$$

at the type $b \rightleftharpoons b$.

□

5

Isomorphic Interpreters from Abstract Machines

In the past chapters, we developed a pure reversible model of computation obtained from type isomorphisms and categorical structures that treats information as a linear resource which can neither be erased nor duplicated. From a programming perspective, our model gives rise to a pure (with no embedded computational effects such as assignments) reversible programming language Π^o based on *partial isomorphisms*. In more detail, in the recursion-free fragment of Π^o, every program witnesses an isomorphism between two finite types; in the full language with recursion, some of these isomorphisms may be partial, i.e., may diverge on some inputs.

In this chapter, we investigate the practicality of writing recursive programs in Π^o. Specifically, we assume we are given some reversible abstract machine and we show how to derive a Π^o program that implements the semantics of the machine. Our derivation is systematic and expressive. In particular, Π^o can handle machines with stuck states because it is based on partial isomorphisms. We illustrate our techniques with simple machines that do bounded iteration on numbers, tree traversals, and a meta-circular interpreter for Π^o. Our choice of starting from reversible abstract machines is supported by the following observations:

- it is an interesting enough class of reversible programs: researchers have invested the effort in manually designing such machines, e.g., the SE(M)CD machine [Kluge, 2000], and the SECD-H [Huelsbergen, 1996];
- every reversible interpreter can be realized as such a machine: this means that our class of programs includes self-interpreters which are arguably a measure of the strength of any reversible language [Axelsen and Glück, 2011, Yokoyama et al., 2008, Yokoyama and Glück, 2007];
- general recursive programs can be systematically transformed to abstract machines: the technique is independent of reversible programs and consists of transforming general recursion to tail recursion and then applying *fission* to split the program into a driver and a small-step machine [Danvy, 2009, Rendel and Ostermann, 2010].

1. Simple Bounded Number Iteration

We illustrate the main concepts and constructions using two simple examples that essentially count n steps. The first machine does nothing else; the second uses this counting ability to add two numbers.

1.1. Counting.

The first machine is defined as follows:

$$\begin{aligned} Numbers, n, m &= 0 \mid n+1 \\ Machine\ states &= \langle n, n \rangle \end{aligned}$$

$$\begin{aligned} Start\ state &= \langle n, 0 \rangle \\ Stop\ State &= \langle 0, n \rangle \end{aligned}$$

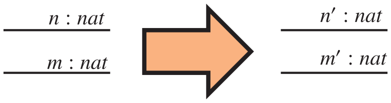
Machine Reductions:

$$\langle n+1, m \rangle \mapsto \langle n, m+1 \rangle$$

The machine is started with a number n in the first position and 0 in the second. Each reduction step decrements the first number and increments the second. The machine stops when the first number reaches 0, thereby taking exactly n steps. For example, if the machine is started in the configuration $\langle 3, 0 \rangle$, it would take exactly 3 steps to reach the final configuration: $\langle 3, 0 \rangle \mapsto \langle 2, 1 \rangle \mapsto \langle 1, 2 \rangle \mapsto \langle 0, 3 \rangle$.

Since the machine transition is clearly reversible, the machine can execute backwards from the final configuration to reach the initial configuration in also 3 steps.

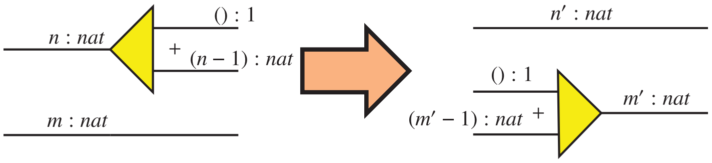
Our goal is to implement this abstract machine in Π^o: We start by writing a combinator $c : nat \times nat \leftrightarrow nat \times nat$ that executes one step of the machine transition. The combinator, when iterated, should map $(n, 0)$ to $(0, n)$ by precisely mimicking the steps of the abstract machine. Let us analyze this desired combinator c in detail starting from its interface:



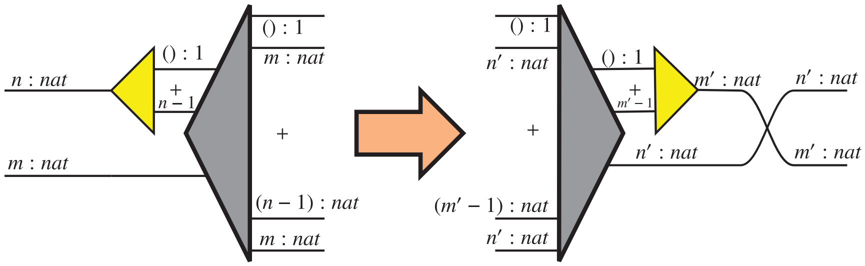
Recall that in Π^o we define nat as the recursive type $\mu x.1+x$ and we have the isomorphisms:

$$fold_{nat} : 1 + nat \leftrightarrow nat : unfold_{nat}$$

The reduction step of the machine examines the first nat and reconstructs the second nat. In other words, the first nat is unfolded to examine its structure and the second nat is folded to reconstruct it:

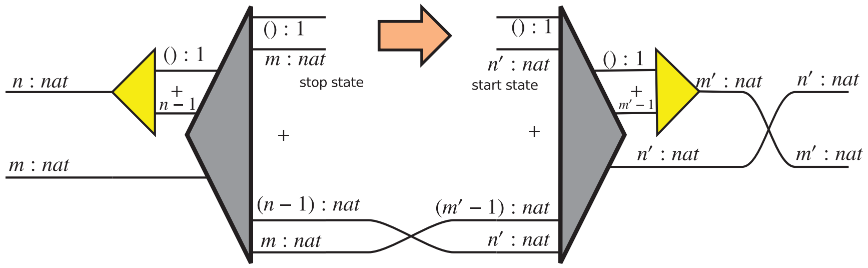


The triangle on the left denotes the $unfold$ isomorphism, which works as follows: If the number n is zero, the output is the top branch which has type 1. If the number was non-zero, the output is on the bottom branch and has value $n - 1$. The triangle on the right is the dual $fold$ isomorphism. We now use distribution to isolate each possible machine state:



More abstractly, we have done the following: for the given machine we have unfolded all the machine components examined by the LHS of the reduction relation $\mapsto$ and dually folded the machine components that have been constructed on the RHS of the reduction relation. The folds and unfolds reveal all the machine states that are considered part of the reduction relation. The next step is to connect related input states to related output, thereby expressing reductions.

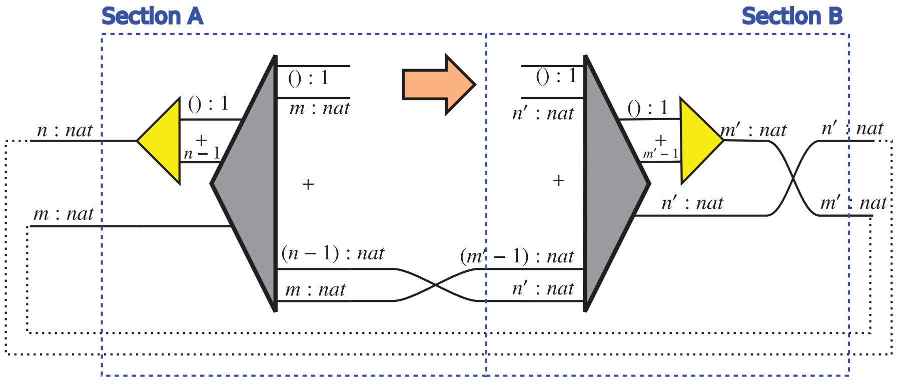
When mapping input state $\langle n, m\rangle$ to output state $\langle n', m'\rangle$, the one reduction relation in the tells us that the equalities $n' = n - 1$ and $m' = m + 1$ must hold. These equalities can be represented by connecting the $n - 1$ input wire to n' on the output side and the m input wire to $m' - 1$ on the output side:



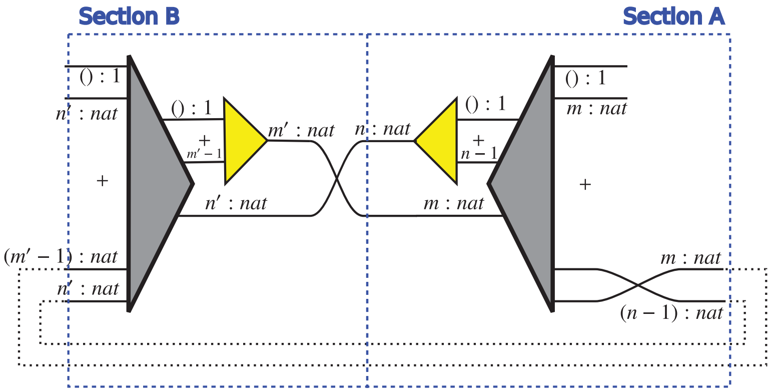
We are now close to the completion of the interpreter. The branch labeled $((), m)$ corresponds to the machine state $\langle 0, m\rangle$ which is the stop state of the machine. Similarly the branch labeled $((), n')$ corresponds to the start state of the machine.

The (incomplete) combinator constructed so far corresponds to a single step of the machine that maps a generic machine state of the form $\langle n, m\rangle$ to another generic machine state. However, we are really interested in an implementation of the machine that maps its start state, of the form $\langle n, 0\rangle$, to the stop state in the course of which the machine may take several reduction steps passing through several intermediate states. The last step in the construction involves introducing a $trace$ operation for iterating the small step realized so far.

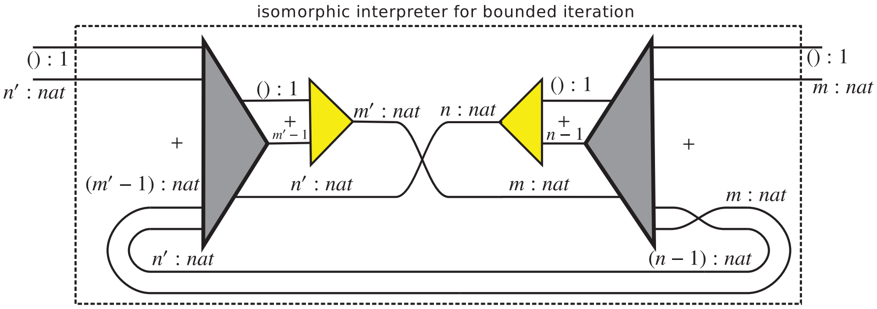
We do this by first "sliding" the two sections of the diagram as shown below:



to



This rearrangement exposes the start and end state of the machine, as show below.



The combinator above is a Π^o implementation of the small step abstract machine. It maps start states of the form $\langle n, 0\rangle$ to stop states of the form $\langle 0, m\rangle$ mimicking the reductions of the original machine. Each iteration of the this combinator executes one machine reduction step. Intermediate machine states, of the form $\langle n, m\rangle$, are the values that are in the feedback loop of the $trace$.

1.2. Adder. Although trivial, the previous example captures the fundamental steps of our construction. For reference, these steps are:

(1) Expand the input to expose enough structure to distinguish the left hand side of each machine state;

(2) Expand the output to expose enough structure to distinguish the right hand side of each machine state;

(3) Shuffle matching input terms to output terms, inserting any appropriate mediating computations.

(4) Slide the two sections to expose the start and stop state and introduce a *trace* to iterate the construction. This is always a mechanical process completely independent of the logic of the machine.

Let us apply these steps to the slightly more interesting example of an adder:

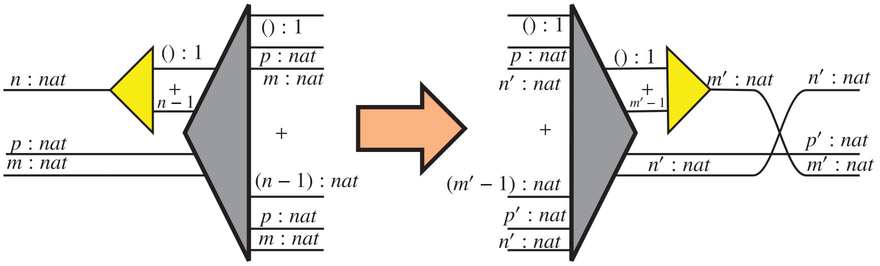
$$\begin{aligned} \mathit{Numbers}, n, m, p &= 0 \mid n+1 \\ \mathit{Machine\ states} &= \langle e, e, e \rangle \end{aligned}$$

$$\begin{aligned} \mathit{Start\ state} &= \langle n, n, 0 \rangle \\ \mathit{Stop\ State} &= \langle 0, n, n \rangle \end{aligned}$$

Machine Reductions:

$$\langle n+1, p, m \rangle \mapsto \langle n, p+1, m+1 \rangle$$

The machine starts with 3 numbers: the two numbers to add and an accumulator initialized to 0. Each step of the machine, decrements one of the numbers and increments the second number and the accumulator. For example, $\langle 3, 4, 0 \rangle \mapsto \langle 2, 5, 1 \rangle \mapsto \langle 1, 6, 2 \rangle \mapsto \langle 0, 7, 3 \rangle$. In general, the sum of the two numbers will be in the second component, and the last component is supposed to record enough information to make the machine reversible. This construction proceeds exactly like the previous one till we have the following situation:

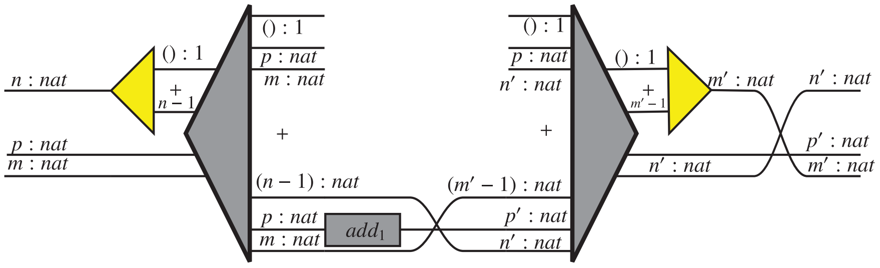


A closer look however reveals that the machine defines a *partial* isomorphism: not all valid final states can be mapped to valid start states. Indeed consider the configuration $\langle 0, 2, 3 \rangle$ which is a valid final state. Going backwards, the transitions start as follows $\langle 0, 2, 3 \rangle \mapsto \langle 1, 1, 2 \rangle \mapsto \langle 2, 0, 1 \rangle$ at which point, the machine gets stuck at a state that is not a valid start state. This is a general problem that we discuss below in detail.

Stuck States. The type systems of most languages are not expressive enough to encode the precise domain and range of a function. For example, in most typed languages, division by zero is considered type-correct and the runtime system is required to deal with such an error. A stuck state of an abstract machine is just a manifestation of this general problem. The common solutions are:

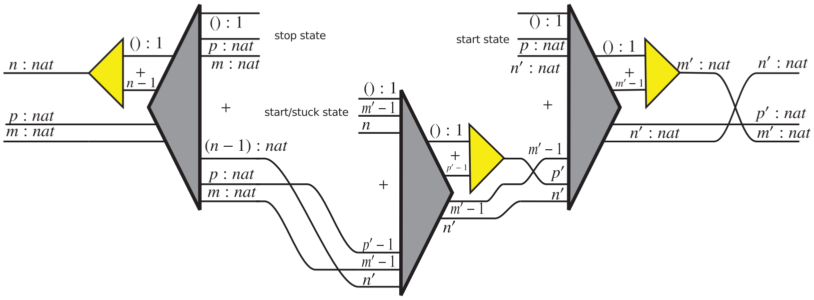
- *Use a more expressive type system.* One could augment Π^o with a richer type system that distinguishes non-zero numbers from those that can be zero, thereby eliminating the $sub_1\ 0$ situation entirely. Similarly, in the meta-circular interpreter in Sec. 4, one could use generalized abstract data types (GADTs) [Cheney and Hinze, 2003, Xi et al., 2003]) to eliminate the stuck states.
- *Diverge.* Another standard approach in dealing with stuck states is to make the machine diverge or leave the output undefined or unobservable in some way.

  If the specific case of the machine above, we can use the primitive $add_1 : nat \leftrightarrow nat$ whose dual $sub_1 : nat \leftrightarrow nat$ is undefined when applied to 0 (see Sec. 3.3). The equalities we would like to express at this point are $n - 1 = n'$, $m = m' - 1$ and that $p + 1 = p'$. We use that add_1 to mediate between p and p':

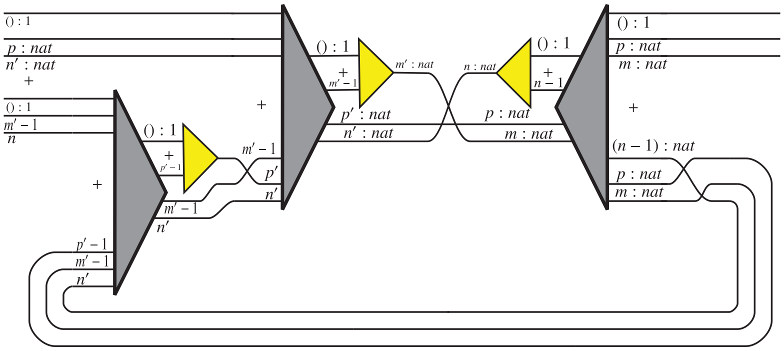
  

  As before, we can 'slide out' the start and stop states and introduce $trace$ for iteration to complete the construction. During reverse execution this Π^o interpreter will diverge on inputs like $\langle 0, 2, 3\rangle$.
- *Stop the machine.* Alternatively, we can consider the stuck state as a valid final state. In the case of the example above, we would treat states of the form $\langle n, 0, n\rangle$ as valid stop states for reverse execution. This gives us two valid start states in

the case of forward execution and makes the isomorphism total. If we chose this approach, the machine would look as shown below by the end of $step$ 3:



Note that there are indeed two "start states" in the interpreter. As before, we can slide the two sides of the diagram and tie the knot using $trace$ to get the desired interpreter:



2. Tree Traversal

We show the generality of our construction by applying it to three non-trivial examples: tree traversal, parity translation and a meta-circular interpreter for Π^o.

The type of binary trees we use is $\mu x.(nat + x \times x)$, i.e., binary trees with no information at the nodes and with natural numbers at the leaves. To define the abstract machine, we need a notion of *tree contexts* to track which subtree is currently being explored. The definitions are shown on the below:

$$\begin{array}{rcl} \mathit{Tree}, t & = & L\ n \mid N\ t\ t \\ \mathit{Tree\ Contexts}, c & = & \Box \mid \mathit{Lft}\ c\ t \mid \mathit{Rgt}\ t\ c \end{array}$$

$$\begin{array}{rcl} \mathit{Machine\ states} & = & \langle t, c\rangle\ \mid \{c, t\} \\ \mathit{Start\ state} & = & \langle t, \Box\rangle \\ \mathit{Stop\ State} & = & \{\Box, t\} \end{array}$$

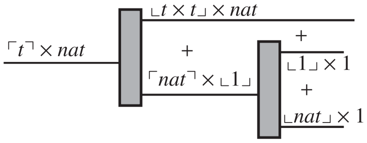
Machine Reductions:

$$\begin{array}{rcl} \langle L\ n, c\rangle & \mapsto & \{c, L\ (n+1)\} \\ \langle N\ t_1\ t_2, c\rangle & \mapsto & \langle t_1, \mathit{Lft}\ c\ t_2\rangle \\ \{\mathit{Lft}\ c\ t_2, t_1\} & \mapsto & \langle t_2, \mathit{Rgt}\ t_1\ c\rangle \\ \{\mathit{Rgt}\ t_1\ c, t_2\} & \mapsto & \{N\ t_1\ t_2, c\} \end{array}$$

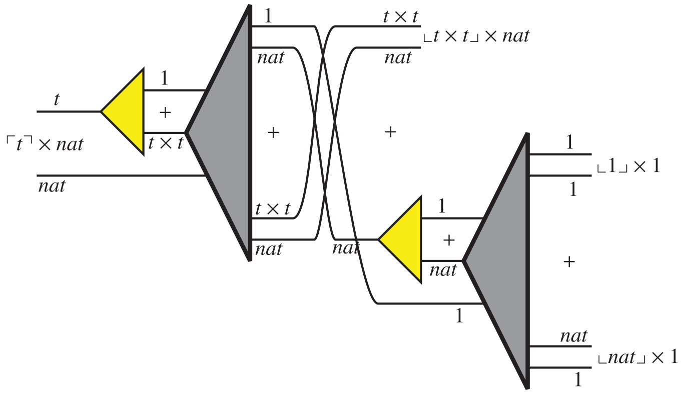
The reduction rules above traverse a given tree and increment every leaf value. The machine here is a little richer than the ones dealt with previously. In particular, we have a two types of machine states – $\langle t, c\rangle$ and $\{t, c\}$. The first of these corresponds to walking down a tree, building up the context in the process. The second corresponds to reconstructing the tree from the context and also switching focus to any unexplored subtrees in the process. There are also two syntactic categories to deal with (trees and tree contexts) where previously we only had numbers. The *fold* and *unfold* isomorphisms that we need for trees and tree contexts are:

$$\begin{array}{rcccl} \mathit{unfold}_t : & t & \leftrightarrow & n + t \times t & : \mathit{fold}_t \\ \mathit{unfold}_c : & c & \leftrightarrow & 1 + c \times t + t \times c & : \mathit{fold}_c \end{array}$$

New Notation. To make the diagrams easier to understand, we introduce a syntactic convenience which combines the first steps of the construction that consist of *fold* / *unfold* and *distribute* / *factor*. We collectively represent these steps using thin vertical rectangle. Also we will introduce the convention that the component that is being expanded (or constructed) will be marked by using $\ulcorner\urcorner$ and the components that are being generated (or consumed) will be marked by $\llcorner\lrcorner$. For example, given a value of type $t \times nat$, the diagram below shows how to first expand the t component and then in one of the generated branches, expand the nat component:



which denotes



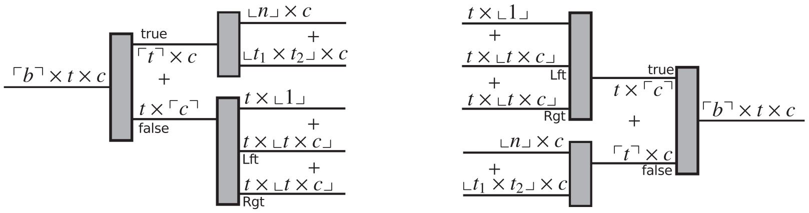
We can now apply our construction. The first step to developing the isomorphic interpreter is to recognize that the two possible kinds of machine states simply hide an implicit $bool = true \mid false$. We make this explicit, by rewriting the abstract machine as a 3-tuple:

$$\begin{aligned} \mathit{Machine\ states} &= \langle bool, t, c\rangle \\ \\ \mathit{Start\ state} &= \langle true, t, \Box\rangle \\ \mathit{Stop\ state} &= \langle false, t, \Box\rangle \end{aligned}$$

Machine Reductions:

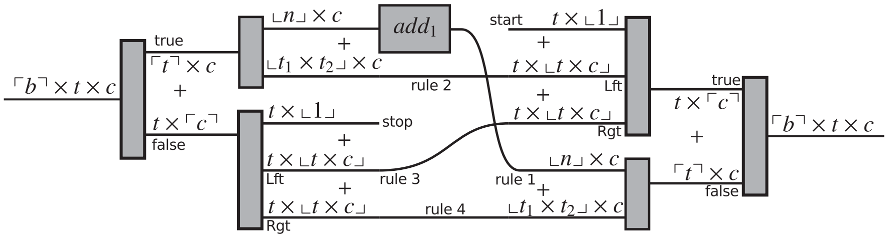
$$\begin{array}{rcll} \langle true, L\ n, c\rangle & \mapsto & \langle false, L\ (n+1), c\rangle & \mathit{rule}\ 1 \\ \langle true, N\ t_1\ t_2, c\rangle & \mapsto & \langle true, t_1, \mathit{Lft}\ c\ t_2\rangle & \mathit{rule}\ 2 \\ \langle false, t_1, \mathit{Lft}\ c\ t_2\rangle & \mapsto & \langle true, t_2, \mathit{Rgt}\ t_1\ c\rangle & \mathit{rule}\ 3 \\ \langle false, t_2, \mathit{Rgt}\ t_1\ c\rangle & \mapsto & \langle false, c, N\ t_1\ t_2\rangle & \mathit{rule}\ 4 \end{array}$$

We start examining the machine components as before:


$\ulcorner b \urcorner \times t \times c$
true
$\ulcorner t \urcorner \times c$
+
$t \times \ulcorner c \urcorner$
false
$\llcorner n \lrcorner \times c$
+
$\llcorner t_1 \times t_2 \lrcorner \times c$
$t \times \llcorner 1 \lrcorner$
+
$t \times \llcorner t \times c \lrcorner$
Lft
+
$t \times \llcorner t \times c \lrcorner$
Rgt
$t \times \llcorner 1 \lrcorner$
+
$t \times \llcorner t \times c \lrcorner$
Lft
+
$t \times \llcorner t \times c \lrcorner$
Rgt
$\llcorner n \lrcorner \times c$
+
$\llcorner t_1 \times t_2 \lrcorner \times c$
true
$t \times \ulcorner c \urcorner$
+
$\ulcorner t \urcorner \times c$
false
$\ulcorner b \urcorner \times t \times c$

On the input side, for $true$ states, we have expanded the tree component and for $false$ states we have expanded the tree contexts. Similarly we have done the opposite on the output side exactly matching up what the abstract machine does. One thing to note is that we dropped the 1 introduced by expanding the $bool$ and instead just labeled the $true$ and $false$ branches. We are ready to start connecting the machine states corresponding to the reductions that we would like:

(1) When we encounter a leaf we would like to increment its value and move to the corresponding $false$ machine state. For the sake of simplicity, in this interpreter we won't be concerned with exposing stuck states: we simply use add_1 whose adjoint sub_1 diverges when applied to 0.
(2) For all the other reduction rules, it is a straightforward mapping of related states following the reduction rules. For readability, we have annotated the diagram below with the names of the reduction rules and we have included subscripts on the various ts to indicate any implicit swaps that should be inserted.


$\ulcorner b \urcorner \times t \times c$
true
$\ulcorner t \urcorner \times c$
+
$t \times \ulcorner c \urcorner$
false
$\llcorner n \lrcorner \times c$
+
$\llcorner t_1 \times t_2 \lrcorner \times c$
add_1
rule 2
$t \times \llcorner 1 \lrcorner$
stop
+
$t \times \llcorner t \times c \lrcorner$
Lft
rule 3
+
$t \times \llcorner t \times c \lrcorner$
Rgt
rule 4
start
$t \times \llcorner 1 \lrcorner$
+
$t \times \llcorner t \times c \lrcorner$
Lft
+
$t \times \llcorner t \times c \lrcorner$
Rgt
$\llcorner n \lrcorner \times c$
rule 1
+
$\llcorner t_1 \times t_2 \lrcorner \times c$
true
$t \times \ulcorner c \urcorner$
+
$\ulcorner t \urcorner \times c$
false
$\ulcorner b \urcorner \times t \times c$

This essentially completes the construction of the (partially) isomorphic tree-traversal interpreter, except for the final step which slides the input and output sides.

3. Parity Translation

While the previous machines expressed traversals of the values with input values reflected in the outputs, the following machine translates one recursive type to another. Consider the type nat (defined to be $\mu x.1 + x$) and its parity encoded version $parity$ (defined to be $\mu x.1 + (1 + x)$). Algebraically, we may write:

$$\begin{array}{rcl} Numbers, n, m & = & 0 \mid n + 1 \\ Parity, par & = & even \mid odd \mid A\ par \end{array}$$

While it is apparent that there is a natural one to one correspondence between their values, it is non-obvious how this can be expressed as a Π^o isomorphism. This map can be expressed in terms of the algebraic type definitions as shown below:

0	$even$
1	odd
2	$A\ even$
3	$A\ odd$
4	$A\ (A\ even)$
...	...

and in terms of their Π^o type encodings as shown below:

$\langle left\ ()\rangle$	$\langle left\ ()\rangle$
$\langle right\ \langle left\ ()\rangle\rangle$	$\langle right\ left\ ()\rangle$
$\langle right\ \langle right\ \langle left\ ()\rangle\rangle\rangle$	$\langle right\ right\ \langle left\ ()\rangle\rangle$
$\langle right\ \langle right\ \langle right\ \langle left\ ()\rangle\rangle\rangle\rangle$	$\langle right\ right\ \langle right\ left\ ()\rangle\rangle$
$\langle right\ \langle right\ \langle right\ \langle right\ \langle left\ ()\rangle\rangle\rangle\rangle\rangle$	$\langle right\ right\ \langle right\ right\ \langle left\ ()\rangle\rangle\rangle$
...	...

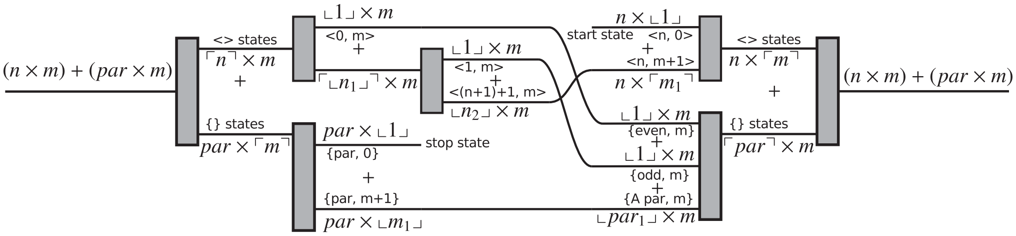
Instead of directly programming in Π^o, one can derive a Π^o combinator for this purpose by first constructing an abstract machine that maps nat to $parity$. A logically reversible abstract machine expressing the translation may be defined as follows:

$$\begin{aligned} Machine\ states &= \langle nat, nat\rangle \mid \{parity, nat\} \\ Start\ state &= \langle n, 0\rangle \\ Stop\ state &= \{par, 0\} \end{aligned}$$

Machine Reductions:

$$\begin{aligned} \langle 0, m\rangle &\mapsto \{even, m\} \\ \langle 1, m\rangle &\mapsto \{odd, m\} \\ \langle (n+1)+1, m\rangle &\mapsto \langle n, m+1\rangle \\ \{par, m+1\} &\mapsto \{A\ par, m\} \end{aligned}$$

The construction of the Π^o combinator is straightforward and proceeds as before. The partial combinator representing the wiring of machine reductions is shown below.


$(n \times m) + (par \times m)$
<> states
$\ulcorner n \urcorner \times m$
+
{} states
$par \times \ulcorner m \urcorner$
$\llcorner 1 \lrcorner \times m$
<0, m>
+
$\ulcorner \llcorner n_1 \lrcorner \urcorner \times m$
$\llcorner 1 \lrcorner \times m$
<1, m>
+
<(n+1)+1, m>
$\llcorner n_2 \lrcorner \times m$
$par \times \llcorner 1 \lrcorner$
{par, 0}
stop state
+
{par, m+1}
$par \times \llcorner m_1 \lrcorner$
$n \times \llcorner 1 \lrcorner$
start state
<n, 0>
+
<n, m+1>
$n \times \ulcorner m_1 \urcorner$
$\llcorner 1 \lrcorner \times m$
{even, m}
+
$\llcorner 1 \lrcorner \times m$
{odd, m}
+
{A par, m}
$\llcorner par_1 \lrcorner \times m$
<> states
$n \times \ulcorner m \urcorner$
+
{} states
$\ulcorner par \urcorner \times m$
$(n \times m) + (par \times m)$

We thank Fritz Henglein for the motivation underlying this example.

4. A Metacircular Interpreter for Π^o

In the final construction we present a metacircular interpreter for Π^o written in Π^o. This is a a non-trivial abstract machine with several cases, but the derivation follows in exactly the same way as before.

4.1. Small-step Abstract Machine for Π^o. Here is a logically reversible small-step abstract machine for Π^o. This presentation is different from that of Def. 2.4 because we are concerned with specifying a small-step abstract machine for translation to Π^o, whereas previously we were satisfied with a natural (big-step) operational semantics.

Definition 4.1 (Syntax of Small-Step Abstract Machine for Π^o).

$$
\begin{array}{rcl}
\mathit{Isomorphisms}, \mathit{iso} & = & \mathit{zeroe} \mid \mathit{zeroi} \mid \mathit{swap}^{+} \mid \mathit{assocl}^{+} \mid \mathit{assocr}^{+} \\
& \mid & \mathit{unite} \mid \mathit{uniti} \mid \mathit{swap}^{\times} \mid \mathit{assocl}^{\times} \mid \mathit{assocr}^{\times} \\
& \mid & \mathit{distrib}_0 \mid \mathit{factor}_0 \mid \mathit{distrib} \mid \mathit{factor} \mid \mathit{fold} \mid \mathit{unfold} \\
\mathit{Combinators}, c & = & \mathit{iso} \mid c \fatsemi c \mid c \times c \mid c + c \mid \mathit{trace}\ c \\
\mathit{Combinator\ Contexts}, \mathit{cc} & = & \Box \mid \mathit{Fst}\ \mathit{cc}\ c \mid \mathit{Snd}\ c\ \mathit{cc} \\
& \mid & \mathit{LeftTimes}\ \mathit{cc}\ c\ v \mid \mathit{RightTimes}\ c\ v\ \mathit{cc} \\
& \mid & \mathit{LeftPlus}\ \mathit{cc}\ c \mid \mathit{RightPlus}\ c\ \mathit{cc} \mid \mathit{Trace}\ \mathit{cc} \\
\mathit{Values}, v & = & () \mid (v, v) \mid L\ v \mid R\ v \\
\\
\mathit{Machine\ states} & = & \langle c, v, \mathit{cc} \rangle \mid \{c, v, \mathit{cc}\} \\
\mathit{Start\ state} & = & \langle c, v, \Box \rangle \\
\mathit{Stop\ State} & = & \{c, v, \Box\}
\end{array}
$$

The data type definitions include syntax for *combinators* and *combinator contexts*. The machine transitions below track the flow of particles through a circuit. The start machine state, $\langle c, v, \Box \rangle$, denotes the particle v about to be evaluated by the circuit c. The end machine state, $[c, v, \Box]$, denotes the situation where the particle v has exited the circuit c.

The interesting thing about the semantics is that it represents a reversible abstract machine. In other words, we can compute the start state from the stop state by changing the reductions $\mapsto$ to run backwards $\leftarrowtail$. When running backwards, we use the isomorphism represented by a combinator c in the reverse direction, i.e., we use the adjoint $c^{\dagger}$.

Definition 4.2 (Reduction rules for Π^o Abstract Machine).

$$
\begin{array}{rcll}
\langle iso, v, cc\rangle & \mapsto & \{iso, iso(v), cc\} & \textit{rule } 1 \\
\langle c_1 \fatsemi c_2, v, cc\rangle & \mapsto & \langle c_1, v, Fst\ cc\ c_2\rangle & \textit{rule } 2 \\
\{c_1, v, Fst\ cc\ c_2\} & \mapsto & \langle c_2, v, Snd\ c_1\ cc\rangle & \textit{rule } 3 \\
\{c_2, v, Snd\ c_1\ cc\} & \mapsto & \{c_1 \fatsemi c_2, v, cc\} & \textit{rule } 4 \\
\langle c_1 + c_2, L\ v, cc\rangle & \mapsto & \langle c_1, v, LeftPlus\ cc\ c_2\rangle & \textit{rule } 5 \\
\{c_1, v, LeftPlus\ cc\ c_2\} & \mapsto & \{c_1 + c_2, L\ v, cc\} & \textit{rule } 6 \\
\langle c_1 + c_2, R\ v, cc\rangle & \mapsto & \langle c_2, v, RightPlus\ c_1\ cc\rangle & \textit{rule } 7 \\
\{c_2, v, RightPlus\ c_1\ cc\} & \mapsto & \{c_1 + c_2, R\ v, cc\} & \textit{rule } 8 \\
\langle c_1 \times c_2, (v_1, v_2), cc\rangle & \mapsto & \langle c_1, v_1, LeftTimes\ cc\ c_2\ v_2\rangle & \textit{rule } 9 \\
\{c_1, v_1, LeftTimes\ cc\ c_2\ v_2\} & \mapsto & \langle c_2, v_2, RightTimes\ c_1\ v_1\ cc\rangle & \textit{rule } 10 \\
\{c_2, v_2, RightTimes\ c_1\ v_1\ cc\} & \mapsto & \{c_1 \times c_2, (v_1, v_2), cc\} & \textit{rule } 11 \\
\langle trace\ c, v, cc\rangle & \mapsto & \langle c, R\ v, Trace\ cc\rangle & \textit{rule } 12 \\
\{c, L\ v, Trace\ cc\} & \mapsto & \langle c, L\ v, Trace\ cc\rangle & \textit{rule } 13 \\
\{c, R\ v, Trace\ cc\} & \mapsto & \{trace\ c, R\ v, cc\} & \textit{rule } 14
\end{array}
$$

Rule (1) describes evaluation by a primitive isomorphism. The definition here shows only the handling of the composition combinators, which are the interesting cases. All the primitive isomorphisms are hidden in the $iso(v)$ application in rule (1). This should be read as "transform v according to the primitive isomorphism iso" and is specified in additive, multiplicative, distributiviy/factoring and folding/unfolding sections of Def. 2.4. Implementing this involves unfolding and distributing the given iso over v and applying the appropriate transformation in each branch. This is straightforward and is hence skipped.

Rules (2), (3) and (4) deal with sequential evaluation. Rule (2) says that for the value v to flow through the sequence $c_1 \fatsemi c_2$, it should first flow through c_1 with c_2 pending in the context ($Fst\ C\ c_2$). Rule (3) says the value v that exits from c_1 should proceed to flow through c_2. Rule (4) says that when the value v exits c_2, it also exits the sequential composition $c_1 \fatsemi c_2$. Rules (5) to (8) deal with $c_1 + c_2$ in the same way. In the case of sums, the shape of the value, i.e., whether it is tagged with $left$ or $right$, determines whether path c_1 or path c_2 is taken. Rules (9), (10) and (11) deal with $c_1 \times c_2$ similarly. In the case of products the value should have the form (v_1, v_2) where v_1 flows through c_1 and v_2 flows through c_2. Both these paths are entirely independent of each other and we could evaluate either one first, or evaluate both in parallel. In this presentation we have chosen to follow c_1 first, but this choice is entirely arbitrary. Rules (12) to (14) deal with trace – rule (12) injects the input value $v : b_2$ into $\mathit{right}\ v_2 : b_1 + b_2$ and rules (13) and (14) determine if iteration should continue or terminate.

Proposition 4.3 (Correspondence)**.** *Evaluation in the small-step evaluator (Def. 4.1 and 4.2) corresponds to evaluation in the natural semantics (Def. 2.4).*

$c\ v \mapsto v'$ *iff* $\langle c, v, \Box \rangle \mapsto^* [c, v', \Box]$

Proof. To prove the above, we first show that a more general lemma holds, namely that: $c\ v \mapsto v'$ iff $\langle c, v, C \rangle \mapsto^* [c, v', C]$.

- To show the left-to-right direction we proceed by induction on the derivation of $c\ v \mapsto v'$. In the case of primitive isomorphisms the condition holds trivially. In the case of composition ($+$, $\times$ and $\fatsemi$), we work out the case of $c_1 + c_2$ in detail. Given $c_1 + c_2\ v \mapsto v'$ we have to show that there is a small step derivation sequence that matches it, i.e. $\langle c_1 + c_2, v, C \rangle \mapsto^* [c_1 + c_2, v', C]$. Here $v : b_1 + b_2$ can be of the form $\mathit{left}\ v_1$ or $\mathit{right}\ v_2$. Assuming $\mathit{left}\ v_1$, we have:

$$\frac{c_1\ v_1 \mapsto v_1' \Longrightarrow \langle c_1, v_1, C' \rangle \mapsto^* \{c_1, v_1', C'\}}{c_1 + c_2\ (\mathit{left}\ v_1) \mapsto \mathit{left}\ v_1' \Longrightarrow ?}$$

  Choosing $C' = \mathit{LeftPlus}\ c_2\ C$, we have the required derivation sequence:
  $\langle c_1 + c_2, \mathit{left}\ v_1, C \rangle \mapsto \langle c_1, v_1, \mathit{LeftPlus}\ c_2\ C \rangle \mapsto^* \{c_1, v_1', \mathit{LeftPlus}\ c_2\ C\} \mapsto \{c_1 + c_2, \mathit{left}\ v_1', C\}$
  The proof follows similarly for the $\mathit{right}\ v_2$ case.
- To show the right-to-left direction we proceed by induction of the sequence of $\mapsto^*$ derivations. Again the case for primitive isomorphisms follows trivially. Taking the case of $c_1 + c_2$ we are required to show that given a sequence $\langle c_1 + c_2, v, C \rangle \mapsto^* \{c_1 + c_2, v', C\}$ there is a derivation tree for $c_1 + c_2\ v \mapsto v'$. In the case that $v : v_1 + b_2$

has the form $\mathit{left}\ v_1$, this follows by observing that the given sequence must have the form $\langle c_1 + c_2, \mathit{left}\ v_1, C\rangle \mapsto \langle c_1, v_1, \mathit{LeftPlus}\ c_2\ C\rangle \mapsto^* \{c_1, v_1', \mathit{LeftPlus}\ c_2\ C\} \mapsto \{c_1{+}c_2, \mathit{left}\ v_1', C\}$. For the strictly smaller inner subsequence by induction we have $c_1\ v_1 \mapsto v_1'$ which lets us complete the derivation of $c_1 + c_2\ (\mathit{left}\ v_1) \mapsto \mathit{left}\ v_1'$. The proof for $\mathit{right}\ v_2$ follows similarly.

- To show the correspondence for the evaluation of trace, i.e. $\mathit{trace}\ c \mapsto v'$ iff $\langle \mathit{trace}\ c, v, C\rangle \mapsto^* \{\mathit{trace}\ c, v', C\}$, we need to first show the following lemma.

  $\mathit{loop}_c\ v \mapsto v'$ iff $\{c, v, \mathit{Trace}\ C\} \mapsto^* \{\mathit{trace}\ c, v', C\}$

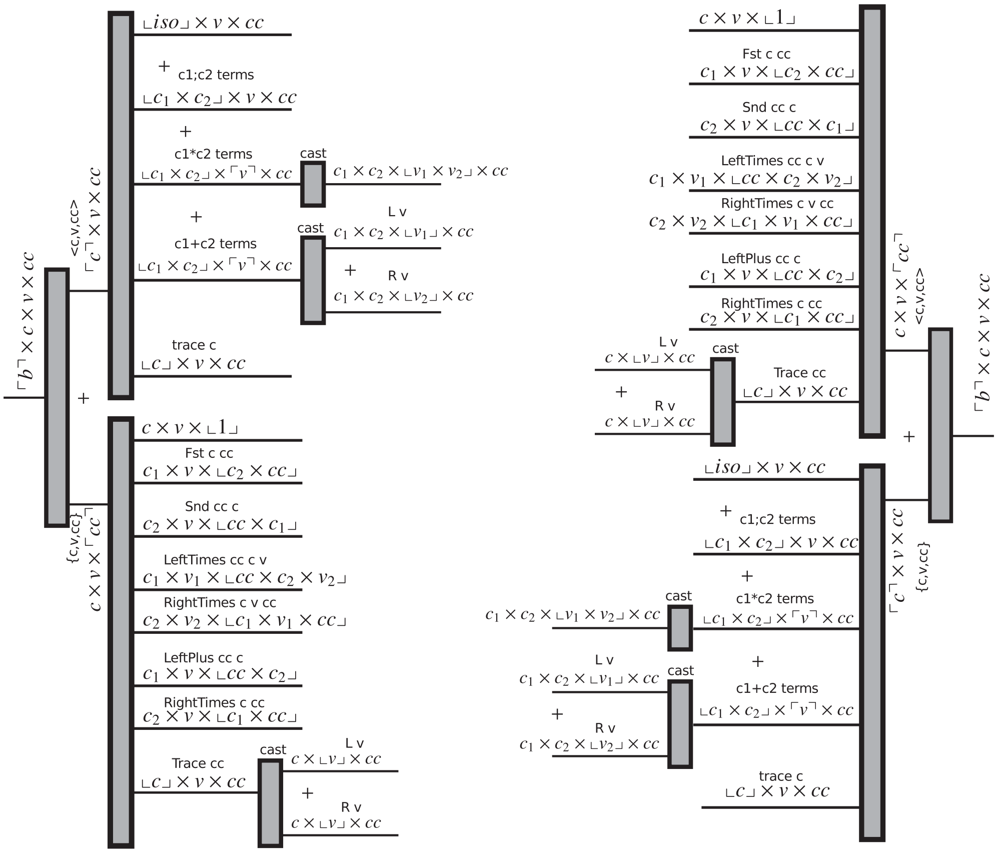
  The left-to-right direction follows from examining the cases of v. When v has the form $\mathit{left}\ v_1$, it follows from the IH on the strictly smaller derivation tree. (Note that we are interested in only the terminating sequences $c\ v \mapsto v'$). When v has the form $\mathit{right}\ v_2$ we terminate with v_2 and we have the reduction sequence $\{c, \mathit{right}\ v_2, \mathit{Trace}\ C\} \mapsto \{\mathit{trace}\ c, v_2, C\}$. The right-to-left direction follows similarly by induction on the length of the reduction sequences.

  Given the above lemma, the required correspondence is readily shown.

$$\frac{\dfrac{c\ (\mathit{right}\ v) \mapsto v' \qquad \dfrac{(\mathit{from}\ \mathit{loop}_c\ \mathit{lemma})}{\mathit{loop}_c\ v' \mapsto v''}}{(c \fatsemi \mathit{loop}_c)\ (\mathit{right}\ v) \mapsto v''}}{(\mathit{trace}\ c)\ v \mapsto v''} \qquad \begin{array}{l} \langle \mathit{trace}\ c, v, C\rangle \mapsto \\ \langle c, \mathit{right}\ v, \mathit{Trace}\ C\rangle \mapsto^* \\ \{c, v', \mathit{Trace}\ C\} \mapsto^* (\mathit{from}\ \mathit{loop}_c\ \mathit{lemma}) \\ \{\mathit{trace}\ c, v'', C\} \end{array}$$

□

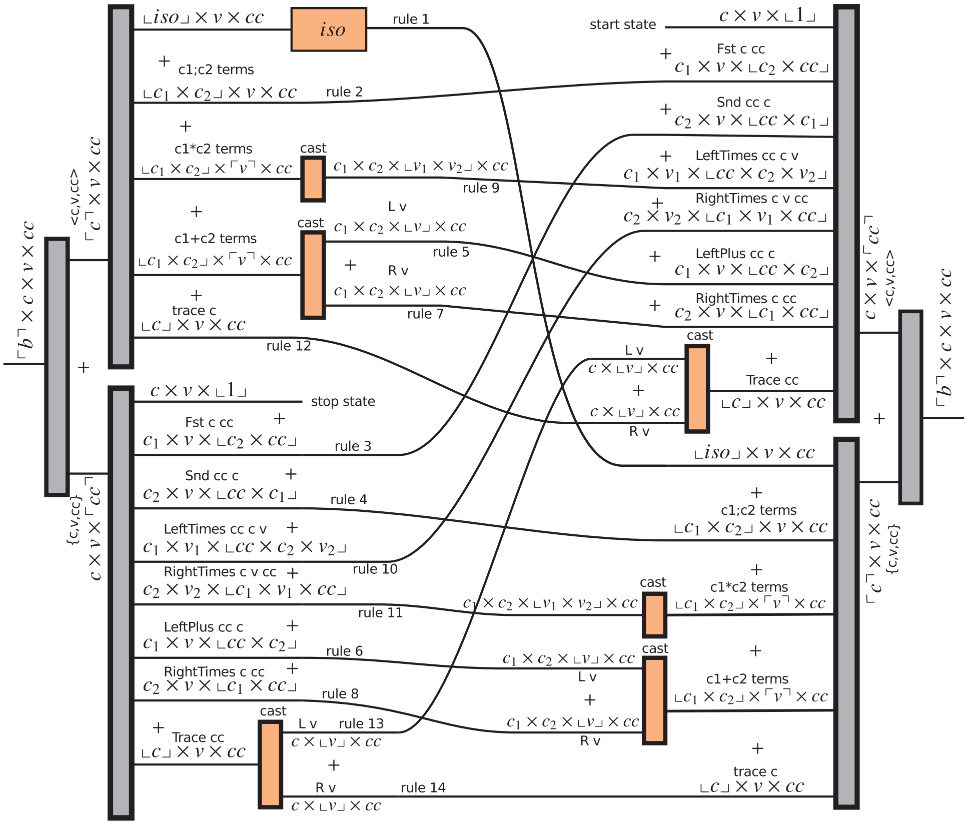
4.2. Deriving a Π^o implementation. We start construction of the Π^o interpreter by examining the machine states, as before:



One the input side, we match against the implicit bool underlying the two kinds of machine states, namely $\langle c, v, cc\rangle$ and $\{c, v, cc\}$. Subsequently we expand out the machine states as required by the reduction rules.

The only subtlety here is with the expected shapes of the values. Stuck states in this machine correspond to runtime values not matching their expected types. In other words, nothing in the machine definition has prevented us from running a *bool* ↔ *bool* type computation with the an input value (). Stated differently, this is just an abstract machine without a type-system. Stuck states of this sort can be handled as described in Sec. 1.2 using GADTs (which eliminates them entirely), divergence, or adding extra halting states. We have abstracted from this choice and marked the relevant cases with combinators labeled $cast$.

To complete the construction we connect the related machine states and label the connections with the appropriate reduction rule number. The application of particular isomorphism to the given value is abstracted into the *iso* combinator below.


$\llcorner iso \lrcorner \times v \times cc$
iso
rule 1
start state
$c \times v \times \llcorner 1 \lrcorner$
c1;c2 terms
$\llcorner c_1 \times c_2 \lrcorner \times v \times cc$
rule 2
Fst c cc
$c_1 \times v \times \llcorner c_2 \times cc \lrcorner$
Snd cc c
$c_2 \times v \times \llcorner cc \times c_1 \lrcorner$
c1*c2 terms
$\llcorner c_1 \times c_2 \lrcorner \times \ulcorner v \urcorner \times cc$
cast
$c_1 \times c_2 \times \llcorner v_1 \times v_2 \lrcorner \times cc$
rule 9
LeftTimes cc c v
$c_1 \times v_1 \times \llcorner cc \times c_2 \times v_2 \lrcorner$
RightTimes c v cc
$c_2 \times v_2 \times \llcorner c_1 \times v_1 \times cc \lrcorner$
c1+c2 terms
$\llcorner c_1 \times c_2 \lrcorner \times \ulcorner v \urcorner \times cc$
L v
$c_1 \times c_2 \times \llcorner v \lrcorner \times cc$
rule 5
R v
rule 7
LeftPlus cc c
$c_1 \times v \times \llcorner cc \times c_2 \lrcorner$
RightTimes c cc
$c_2 \times v \times \llcorner c_1 \times cc \lrcorner$
trace c
$\llcorner c \lrcorner \times v \times cc$
rule 12
$\ulcorner b \urcorner \times c \times v \times cc$
<c,v,cc>
$\ulcorner c \urcorner \times v \times cc$
$c \times v \times \llcorner 1 \lrcorner$
stop state
$c \times \llcorner v \lrcorner \times cc$
Trace cc
$\llcorner c \lrcorner \times v \times cc$
rule 3
rule 4
$\llcorner iso \lrcorner \times v \times cc$
rule 10
rule 11
rule 6
rule 8
rule 13
rule 14
{c,v,cc}
$c \times v \times \ulcorner cc \urcorner$
$c \times v \times \ulcorner cc \urcorner$
$\ulcorner c \urcorner \times v \times cc$
+

While the other abstract machines we have dealt with thus far were all terminating machines, except for the handling of stuck states, this construction here may also diverge if the input combinator c diverges on input value v. The final step that introduces the *trace* is entirely mechanical and has been skipped.

6

Information Effects

This chapter establishes the notion of *information effects*. Information effects are the gain or loss of information by a computation and are closely tied to the relationship between logical reversibility and information preservation. Information effects are the *cause* for irreversibility in computation. Early versions of this idea can be traced back to Toffoli [1980], Fredkin and Toffoli [1982] and others [Bennett and Landauer, 1985, Bennett, 1973, Landauer, 1961].

We establish that information effects are a form of computational effects – albeit a very fine-grained form of the same – and can be encapsulated and studied using mechanisms developed to study other computational effects. We encapsulate information effects using an *arrow metalanguage*. By encapsulating information effects in this way, we show that the gap between conventional irreversible computation and reversible computation can be captured by a type-and-effect system.

The interesting consequence of adding information effects is that we can now express maps from $b_1 \times b_2$ to b_1 (and to b_2), i.e. we can express projections, and maps from b_1 (or from b_2) to $b_1 + b_2$, i.e. we can express injections. In other words, the multiplicative monoidal tensor behaves like categorical products in the presence on information effects and the additive monoidal tensor behaves like a co-product.

1. Information and Logical Reversibility

1.1. Thermodynamics of Computation and Information. Early work into the thought experiment known as "Maxwell's demon" by Landauer [1961] and Bennett [1973] established the remarkable result, known as the *Landauer principle*, relating irreversible computations to increase in information uncertainty (entropy). Logically reversible computations are information-preserving, thereby conserving the entropy during computation[Malacaria and Smeraldi, 2012]. The following definitions establish the notion of information-preservation.

Definition 1.1 (Entropy of a variable). *Let 'b' be a (not necessarily finite) type whose values are labeled $b^1, b^2, \ldots$. Let ξ be a random variable of type b that is equal to b^i with probability p_i. The entropy of ξ is defined as $-\sum p_i \log p_i$.*

For example, consider a variable ξ of type $bool \times bool$. The information content of this variable depends on the probability distribution of the four possible $bool \times bool$ values. If we have a computational situation in which the pair $(false, false)$ could occur with probability $1/2$, the pairs $(false, true)$ and $(true, false)$ can each occur with probability $1/4$, and the pair $(true, true)$ cannot occur, the information content of ξ would be:

$$1/2 \log 2 + 1/4 \log 4 + 1/4 \log 4 + 0 \log 0$$

which equals 1.5 bits of information. If, however, the four possible pairs had an equal probability, the same formula would calculate the information content to be 2 bits, which is the maximal amount for a variable of type $bool \times bool$. The minimum entropy 0 corresponds to a variable that happens to be constant with no uncertainty.

Definition 1.2 (Output entropy of a function)**.** *Consider a function $f : b_1 \to b_2$ where b_2 is a (not necessarily finite) type whose values are labeled $b_2^1, b_2^2, \ldots$. The output entropy of the function is given by $-\sum q_j \log q_j$ where q_j indicates the probability of the output of the function to have value ${b_2}^j$.*

Definition 1.3. *We say a function is* information-preserving *if its output entropy is equal to the entropy of its input.*

Let us consider some examples.

(1) Consider the $bool \to bool$ function not. Let p_F and p_T be the probabilities that the input is $false$ or $true$ respectively. The outputs occur with the reverse probabilities, i.e., p_T is the probability that the output is $false$ and p_F is the probability that the output is $true$. Hence the output entropy of the function is $-p_F \log p_F - p_T \log p_T$ which is the same as the input entropy and the function is information-preserving.

(2) As another example, consider the $bool \to bool$ function $constT(x) = true$ which discards its input. The output of the function is always $true$ with no uncertainty, which means that the output entropy is 0, and that the function is not information-preserving. As a third example, consider the function and and let the inputs occur with equal probabilities, i.e., let the entropy of the input be 2. The output is $false$

with probability $3/4$ and $true$ with probability $1/4$, which means that the output entropy is about 0.8 and the function is not information-preserving.

(3) As a final example, consider the $bool \rightarrow bool \times bool$ function $fanout\ (x) = (x, x)$ which duplicates its input. Let the input be $false$ with probability p_F and $true$ be probability p_T. The output is $(false, false)$ with probability p_F and $(true, true)$ with probability p_T which means that the output entropy is the same as the input entropy and the function is information-preserving.

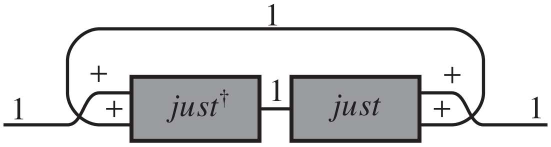
1.2. Logical Reversibility. We are now ready to formalize the connection between reversibility and entropy, once we define logical reversibility of computations.

Definition 1.4 (Logical reversibility [Zuliani, 2001]). *A function $f : b_1 \rightarrow b_2$ is logically reversible if there exists an inverse function $g : b_2 \rightarrow b_1$ such that for all values $v_1 \in b_1$ and $v_2 \in b_2$, we have: $f(v_1) = v_2$ iff $g(v_2) = v_1$.*

The main proposition that motivates and justifies our approach is that logically reversible functions are information-preserving.

Proposition 1.5. *A function is logically reversible iff it is information-preserving.*

Looking at the examples above, we argued that $constT$, and are not information-preserving and that not, $fanout$ are information-preserving. As expected, neither $constT$ nor and are logically reversible and not is logically reversible. The situation with $fanout$ is however subtle and deserves some explanation. First, note that the definition of logical reversibility does not require the functions to be total, and hence it is possible to define a *partial* function $fanin$ that is the logical inverse of $fanout$. The function $fanin$ maps $(false, false)$ to $false$, $(true, true)$ to $true$ and is undefined otherwise. Arguing that partial functions like $fanin$ are information-preserving requires some care. Let the inputs to $fanin$ occur with equal probabilities, i.e., let the entropy of the input be 2. Disregarding the partiality of $fanin$, one might reason that the output is $false$ with probability $1/4$ and $true$ with probability $1/4$ and hence that the output entropy is 1 which contradicts the fact that $fanin$ is logically reversible. The subtlety is that entropy is defined with respect to observing some probabilistic event: an infinite loop is not an event that can be observed and hence the entropy analysis, just like the definition of logical reversibility, only applies to the pairs of inputs and outputs on which the function is defined. In the case of $fanin$ this means that the only inputs that can be considered are $(\mathit{false}, \mathit{false})$ and $(\mathit{true}, \mathit{true})$ and in this case it is clear that the function is information-preserving as expected. In fact, non-termination is entirely independent of information content as can be seen in the infinite loop below of type $1 \rightleftharpoons 1$ that takes 0 bits of information and produces 0 bits of information.



1.3. Linear Logic. Linear logic [Girard, 1987] is often used as a framework for controlling resource use. Linearity however must not be confused with the criterion of information preservation presented here. Consider $constT'(x) = if\ x\ then\ true\ else\ true$ which is extensionally equivalent to the constant function $constT(x) = true$ above. In a linear type system that tracks the *syntactic* occurrences of variables, $constT'$ would be deemed acceptable because x is linearly used. However as shown above the function $constT$ is not information-preserving. Despite this difference, there does however appear to be some deep connections between linear logic and the physical notions of reversible and quantum computing. Indeed as Girard explains [Girard, 1987, pp. 6,17], linear logic embodies a simple and radical change of viewpoint from other logics and this change has a physical flavor.

2. Arrow metalanguage over Π

To construct our arrow metalanguage ML_Π, we simply add the generic arrow combinators to Π and add the particular operators that model the information effects we wish to model. This metalanguage isolates the typing and semantics of the effects from the remainder of the language, thus playing a role similar to Moggi's monadic metalanguage [Moggi, 1991] in the translation of conventional computational effects (e.g., control operators [Hatcliff and Danvy, 1994]). The metalanguage is defined, not by extending the λ-calculus with monadic combinators, but by extending Π/Π^o with arrow combinators. In our setting, Π/Π^o plays the role of the pure effect-free language, just like the λ-calculus plays the role of the pure effect-free language in the conventional setting.

Traditional Arrows. We give a brief introduction to traditional arrows to help explain our design of our arrow metalanguage. Like monads, traditional arrows have been defined in the context of the lambda calculus [Hughes, 2000]. Traditional arrows require the addition of three operators, `arr`, $\ggg$ and `first` and one parametric type, $t_1 \leadsto t_2$, to the typed λ-calculus. When dealing with sum types, a fourth operator called `left` is usually included. The following is the syntax of a simply typed λ-calculus extended with an arrow structure.

$$\begin{array}{rcl} types, t & = & base\ types \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \mid t_1 \leadsto t_2 \\ expressions, e & = & x \mid \lambda x.e \mid e_1\ e_2 \\ & \mid & \texttt{arr}\ e \mid e_1 \ggg e_2 \mid \texttt{first}\ e \mid \texttt{left}\ e \end{array}$$

The type $t_1 \leadsto t_2$ denotes the type of effectful terms. The above syntax does not include any effect operators just yet. When discussing a particular effect, one would complete the metalanguage above by including the particular effect operators. The arrow terms have the following type judgments:

$$\frac{\Gamma \vdash e : t_1 \rightarrow t_2}{\Gamma \vdash \texttt{arr}\ e : t_1 \leadsto t_2} \quad \frac{\Gamma \vdash e_1 : t_1 \leadsto t_2 \quad \Gamma \vdash e_2 : t_2 \leadsto t_3}{\Gamma \vdash (e_1 \ggg e_2) : t_1 \leadsto t_3}$$

$$\frac{\Gamma \vdash e : t_1 \leadsto t_2}{\Gamma \vdash \texttt{first}\ e : t_1 \times t_3 \leadsto t_2 \times t_3} \quad \frac{\Gamma \vdash e : t_1 \leadsto t_2}{\Gamma \vdash \texttt{left}\ e : t_1 + t_3 \leadsto t_2 + t_3}$$

Our Arrow Metalanguage. To construct our arrow metalanguage we add the three basic arrow operators and *left* (since our language has sums) to our base language, Π. The resulting syntax of ML_Π is shown below:

$$\begin{array}{rcl} types, t & ::= & b \leftrightarrow b \mid b \leadsto b \\ arrow\ computations, a & ::= & iso \mid a + a \mid a \times a \mid a \fatsemi a \\ & \mid & \texttt{arr}\ a \mid a \ggg a \\ & \mid & \texttt{first}\ a \mid \texttt{left}\ a \end{array}$$

The types of these operators in ML_Π mimic their original types in $\lambda^{\rightarrow\times+}$.

$$\frac{a : b_1 \leftrightarrow b_2}{\texttt{arr}\ a : b_1 \leadsto b_2} \quad \frac{a_1 : b_1 \leadsto b_2 \quad a_2 : b_2 \leadsto b_3}{a_1 \ggg a_2 : b_1 \leadsto b_3}$$

$$\frac{a : b_1 \leadsto b_2}{\texttt{first}\ a : b_1 \times b_3 \leadsto b_2 \times b_3} \quad \frac{a : b_1 \leadsto b_2}{\texttt{left}\ a : b_1 + b_3 \leadsto b_2 + b_3}$$

We can encode several other handy operators in terms of these:

- $\texttt{second}\ a = (\texttt{arr}\ \mathit{swap}^{\times}) \ggg \texttt{first}\ a \ggg (\texttt{arr}\ \mathit{swap}^{\times})$
- $a_1 \otimes a_2 = \texttt{first}\ a_1 \ggg \texttt{second}\ a_2$
- $\texttt{right}\ a = (\texttt{arr}\ \mathit{swap}^{+}) \ggg \texttt{left}\ a \ggg (\texttt{arr}\ \mathit{swap}^{+})$
- $a_1 \oplus a_2 = \texttt{left}\ a_1 \ggg \texttt{right}\ a_2$

These have the type judgments:

$$\frac{a : b_2 \leadsto b_3}{\texttt{second}\ a : b_1 \times b_2 \leadsto b_1 \times b_3} \quad \frac{a_1 : b_1 \leadsto b_2 \quad a_2 : b_3 \leadsto b_4}{a_1 \otimes a_2 : b_1 \times b_3 \leadsto b_2 \times b_4}$$

$$\frac{a : b_2 \leadsto b_3}{\texttt{right}\ a : b_1 + b_2 \leadsto b_1 + b_3} \quad \frac{a_1 : b_1 \leadsto b_2 \quad a_2 : b_3 \leadsto b_4}{a_1 \oplus a_2 : b_1 + b_3 \leadsto b_2 + b_4}$$

Given these derived operators, it easy to characterize what the arrow structure has achieved in Π. The operator `arr` lifts $\leftrightarrow$ types to $\leadsto$ types, i.e. it allows us to treat pure $b_1 \leftrightarrow b_2$ operations as if they were effectful operations of type $b_1 \leadsto b_2$. The operator $\ggg$ is the lifted form of $\fatsemi$, and allows us to sequentially compose effectful operations. The operator $\otimes$ is the lifted form of $\times$ and allows multiplicative parallel composition of effectful combinators. Operators `first` and `second` are simply restricted forms of $\otimes$. The operator $\oplus$ plays the similar role for the additive monoid.

An important feature of the arrow metalanguage as constructed so far is that all the types related by $\leadsto$ are still isomorphic. The introduction of the arrow operators by themselves does not admit any non-isomorphic transformations. These are relegated to particular concrete arrow we build over the metalanguage by adding effects. The analogous situation exists with monads, where the MML by itself does not force one to commit to effects of `State` or `Cont` – these effects are only defined by the particular monad.

2.1. Arrow laws. Hughes [2000] prescribed 9 “laws” that any concrete realization of the arrow operators should follow. Later work [Lindley et al., 2010] showed that one of these laws is redundant. Since these laws were originally prescribed in the setting of the λ-calculus, we have adapted them to Π by replacing λ-calculus terms in the original laws by equivalent Π ones.

Of the remaining 8 Hughes arrow laws, we have dropped one law and added 4 more. The dropped law (shown below) requires a non-isomorphic operation $\texttt{fstA} : b_1 \times b_2 \rightsquigarrow b_1$. Such an operation cannot be admitted into the metalanguage that allows only isomorphic operations. Hughes [2000] does not provide laws for the additive fragment of the arrow combinators and hence 4 laws have been added to closely match the laws for the multiplicative fragment.

As with the original arrow laws, these arrow laws should be obeyed by any concrete implementation of the metalanguage over Π/Π^o.

Definition 2.1. *Arrow Laws for* Π

Laws for sequencing:

$$\begin{aligned}
\texttt{arr}\ id \ggg f &= f && (1)\\
(f \ggg g) \ggg h &= f \ggg (g \ggg h) && (2)\\
\texttt{arr}\ (f \fatsemi g) &= \texttt{arr}\ f \ggg \texttt{arr}\ g && (3)
\end{aligned}$$

Laws for pairs:

$$\begin{aligned}
\texttt{first}\ (\texttt{arr}\ f) &= \texttt{arr}\ (f \times id) && (4)\\
\texttt{first}\ (f \ggg g) &= \texttt{first}\ f \ggg \texttt{first}\ g && (5)\\
\texttt{first}\ f \ggg \texttt{arr}\ (id \times g) &= \texttt{arr}\ (id \times g) \ggg \texttt{first}\ f && (6)\\
\texttt{first}\ (\texttt{first}\ f) \ggg \texttt{arr}\ assocr^{\times} &= \texttt{arr}\ assocr^{\times} \ggg \texttt{first}\ f && (7)
\end{aligned}$$

Laws for sums:

$$\begin{aligned}
\texttt{left}\ (\texttt{arr}\ f) &= \texttt{arr}\ (f + id) && (4')\\
\texttt{left}\ (f \ggg g) &= \texttt{left}\ f \ggg \texttt{left}\ g && (5')\\
\texttt{left}\ f \ggg \texttt{arr}\ (id + g) &= \texttt{arr}\ (id + g) \ggg \texttt{left}\ f && (6')\\
\texttt{left}\ (\texttt{left}\ f) \ggg \texttt{arr}\ assocr^{+} &= \texttt{arr}\ assocr^{+} \ggg \texttt{left}\ f && (7')
\end{aligned}$$

Finally, this is the dropped law:

$\mathit{first}\ f \ggg \texttt{fstA} = \texttt{fstA} \ggg f$

3. Information Effects : `create` and `erase`

To construct our arrow metalanguage ML_Π, we simply add the generic arrow combinators to Π and add the particular operators that model the information effects we wish to model. In particular, we add two operators `create` and `erase` which correspond to the creation and erasure of information. These operators are not isomorphisms and hence cannot have $\leftrightarrow$ types. They can only have arrow types.

Definition 3.1. *(Syntax of ML_Π) The sets of value types, values, and isomorphisms are identical to the corresponding sets in Π (see Def. 2.3). The extended combinator types and arrow computations are defined as follows:*

$$\begin{array}{rcl} \mathit{types}, t & ::= & b \leftrightarrow b \mid b \rightsquigarrow b \\ \mathit{arrow\ comp.}, a & ::= & \mathit{iso} \mid a + a \mid a \times a \mid a \fatsemi a \\ & \mid & \texttt{arr}\ a \mid a \ggg a \mid \texttt{first}\ a \mid \texttt{left}\ a \\ & \mid & \texttt{create}_b \mid \texttt{erase} \end{array}$$

The type $b_1 \rightsquigarrow b_2$ is our notion of arrows. The three operations `arr`, $\ggg$, and `first` are essential for any notion of arrows. The operation `left` is needed for arrows that also implement some form of choice. The two operators $\texttt{create}_b$ and `erase` model the particular effects in the information metalanguage. The types of the arrow combinators in ML_Π are similar to their original types in the traditional arrow calculus except that `arr` lifts $\leftrightarrow$ types to the abstract arrow type $\rightsquigarrow$ instead of lifting regular function types to the abstract arrow type:

Definition 3.2. *Type system for ML_Π*

$$\frac{a : b_1 \leftrightarrow b_2}{\texttt{arr}\ a : b_1 \rightsquigarrow b_2} \qquad \frac{a_1 : b_1 \rightsquigarrow b_2 \quad a_2 : b_2 \rightsquigarrow b_3}{a_1 \ggg a_2 : b_1 \rightsquigarrow b_3}$$

$$\frac{a : b_1 \rightsquigarrow b_2}{\texttt{first}\ a : b_1 \times b_3 \rightsquigarrow b_2 \times b_3} \qquad \frac{a : b_1 \rightsquigarrow b_2}{\texttt{left}\ a : b_1 + b_3 \rightsquigarrow b_2 + b_3}$$

$$\frac{}{\texttt{create}_b : 1 \rightsquigarrow b} \qquad \frac{}{\texttt{erase} : b \rightsquigarrow 1}$$

The semantics is specified using the relation $\mapsto_{ML}$ which refers to the reduction relation $\mapsto$ for Π. We only present the reductions for the arrow constructs. The reductions for the pure Π combinators remain unchanged.

Definition 3.3. *Operational Semantics of ML*$_\Pi$

$$\frac{a\ v_1 \mapsto v_2}{(\texttt{arr}\ a)\ v_1 \mapsto_{ML} v_2} \quad \frac{a_1\ v_1 \mapsto_{ML} v_2 \quad a_2\ v_2 \mapsto_{ML} v_3}{(a_1 \ggg a_2)\ v_1 \mapsto_{ML} v_3}$$

$$\frac{a\ v_1 \mapsto_{ML} v_2}{(\texttt{first}\ a)\ (v_1, v_3) \mapsto_{ML} (v_2, v_3)}$$

$$\frac{a\ v_1 \mapsto_{ML} v_2}{(\texttt{left}\ a)\ (\mathit{left}\ v_1) \mapsto_{ML} \mathit{left}\ v_2} \quad \frac{}{(\texttt{left}\ a)\ (\mathit{right}\ v) \mapsto_{ML} \mathit{right}\ v}$$

$$\frac{}{\texttt{create}_b() \mapsto_{ML} \phi(b)} \quad \frac{}{\texttt{erase}\ v \mapsto_{ML} ()}$$

The operator `erase` at type b takes any value of type b and returns $()$ which contains no information. For any type b, $\texttt{create}_b$ returns a fixed (but arbitrary) value of type b which we call $\phi(b)$, i.e. the least value of a type (Chapter 3 Def. 1.4).

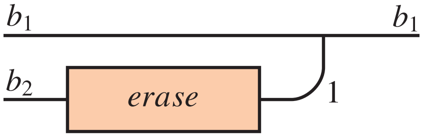
The two operators $\texttt{create}_b$ and `erase`, along with the structure provided by the arrow metalanguage, are expressive enough to implement a number of interesting idioms. In particular, it is possible to erase a part of a data structure (as shown using `fstA` below); it is possible to inject a value in a sum type (as shown using `leftA` below); it is possible to forget about choices (as shown using join below); and it is possible to make a copy of any value (as shown using clone below).

Proposition 3.4. *ML*$_\Pi$ *respects the arrow laws*

PROOF. The proof for this is straightforward. We use operational equivalence as our notion of equality. i.e. $c_1 = c_2$ iff for all v, $c_1\ v \mapsto v'$ iff $c_2\ v \mapsto v'$. Since the arrow laws behave extensionally with respect to values, these are easy to check using the $\mapsto_{ML}$ reductions defined above. □

4. Expressiveness of ML$_\Pi$

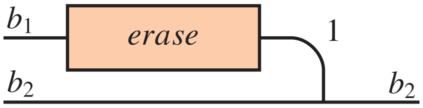
4.1. Erasing part of a data structure (`fstA`). The combinator `fstA` of type $b_1 \times b_2 \rightsquigarrow b_1$ takes a pair and erases the second component. We apply `erase` to the second component of the pair and then appeal to $(\texttt{arr}\ \mathit{unite}) : 1 \times b \rightsquigarrow b$ to absorb the $()$. Thus we have:



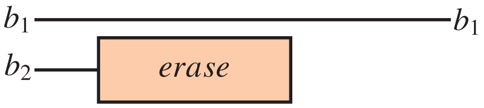
$\texttt{fstA} : b_1 \times b_2 \rightsquigarrow b_2$

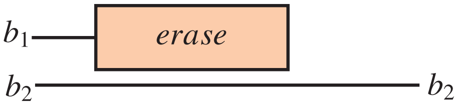
$\texttt{fstA} = \texttt{second erase} \ggg \texttt{arr}\ (\mathit{swap}^{\times} \mathbin{⨾} \mathit{unite})$

where $\texttt{second}\ a = (\texttt{arr}\ \mathit{swap}^{\times}) \ggg \texttt{first}\ a \ggg (\texttt{arr}\ \mathit{swap}^{\times})$. The combinator `sndA` that deletes the first component of a pair is defined symmetrically.



Often we drop the wire for the type 1 and simply draw them as shown below.

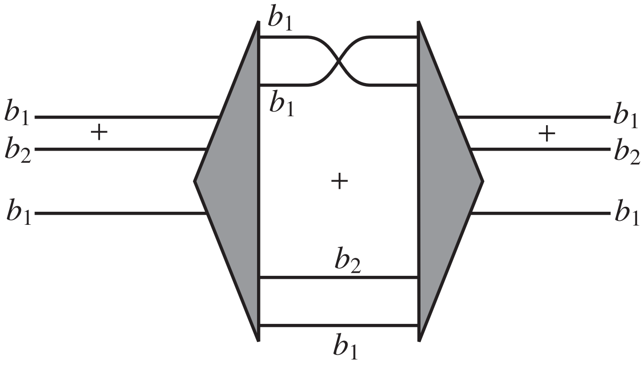




4.2. Injecting a value in a larger type (`leftA`). The combinator $\texttt{leftA} : b_1 \rightsquigarrow b_1 + b_2$ takes a value and injects it in a larger sum type. Its definition is involved so we begin by defining the following combinator in Π:

$\mathit{leftSwap} : (b_1 + b_2) \times b_1 \leftrightarrow (b_1 + b_2) \times b_1$

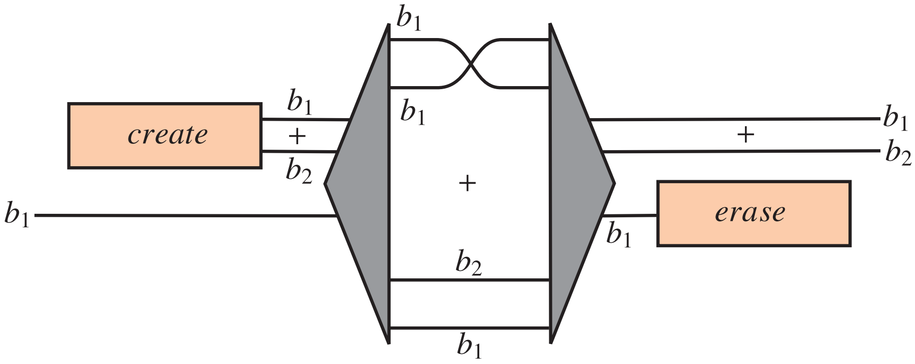
$\mathit{leftSwap} = (\mathit{distrib} \mathbin{⨾} (\mathit{swap}^{\times} + \mathit{id}) \mathbin{⨾} \mathit{factor})$



Given values v_1 and v_1' both of type b_1, the application of $\mathit{leftSwap}$ to $(\mathit{left}\ v_1, v_1')$ produces $(\mathit{left}\ v_1', v_1)$, which moves the left constructor from v_1 to v_1'. We use this combinator to implement `leftA` as follows. Let us give the input to `leftA` the name v_1'. We first create a default value *left* v_1 of type $b_1 + b_2$, use *leftSwap* to produce $(left\ v_1', v_1)$, and complete the definition by using `fstA` to erase the default value v_1. Thus `leftA` is:

$(\texttt{arr}\ uniti) \ggg (\texttt{first create}_{b_1+b_2}) \ggg (\texttt{arr}\ leftSwap) \ggg \texttt{fstA}$

or pictorially:



The symmetric combinator `rightA` can be defined similarly.

4.3. Forgetting about choices (*join*). We define an operator $join : b+b \rightsquigarrow b$ that takes a value of type b tagged by either *left* and *right* and removes the tag. The definition converts the input $b + b$ to $(1 + 1) \times b$ and then erases the first component:

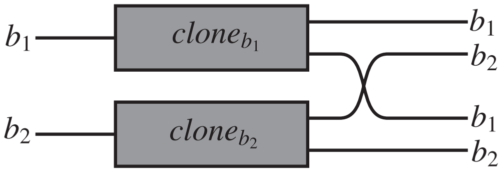
$join = ((\texttt{arr}\ uniti) \oplus (\texttt{arr}\ uniti)) \ggg (\texttt{arr}\ factor) \ggg \texttt{sndA}$

4.4. Copying values ($clone_b$). As the following lemma shows, it is possible, for any type b, to define an operator $clone_b$ that can be used to copy values of that type.

Lemma 4.1 (Cloning). *For any type b, we can construct an operator $clone$ of type $b \rightsquigarrow b \times b$ such that: $clone\ v \mapsto_{ML} (v, v)$*

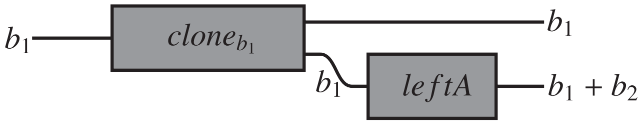
PROOF. We proceed by induction on the type b:

- Case 1: We need to exhibit a combinator $a : 1 \rightsquigarrow 1 \times 1$ and this is given by $a = \texttt{arr}\ uniti$
- Case $b_1 \times b_2$: By induction we have combinators $a_1 : b_1 \rightsquigarrow b_1 \times b_1$ and $a_2 : b_2 \rightsquigarrow b_2 \times b_2$ and we have to construct $a : (b_1 \times b_2) \rightsquigarrow (b_1 \times b_2) \times (b_1 \times b_2)$. The required combinator a uses a_1 and a_2 to clone the components and then shuffles the pairs into place:

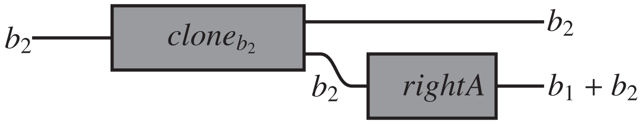


Thus we can write $a = (a_1 \otimes a_2) \ggg \texttt{arr}\ \mathit{shuffle}$ where $\mathit{shuffle}$ is $assocl^{\times} \fatsemi (assocr^{\times} \times id) \fatsemi (id \times swap^{\times} \times id) \fatsemi (assocl^{\times} \times id) \fatsemi assocr^{\times}$.

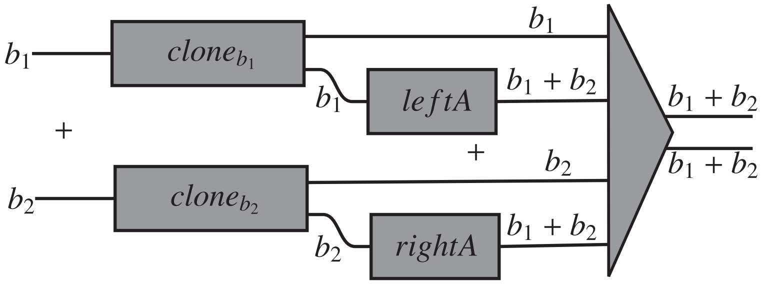
- Case $b_1 + b_2$: By induction we have combinators $a_1 : b_1 \leadsto b_1 \times b_1$ and $a_2 : b_2 \leadsto b_2 \times b_2$ we have to construct $a : (b_1 + b_2) \leadsto (b_1 + b_2) \times (b_1 + b_2)$. Consider the diagram below which has the type $b_1 \leadsto b_1 \times (b_1 + b_2)$:



This combinator clones b_1 and then applies `leftA` to one of the copies. Let us call this combinator $a'_1 : b_1 \leadsto b_1 \times (b_1 + b_2)$. We can do the same with b_2, except that we apply `rightA`, resulting in combinator $a'_2 : b_2 \leadsto b_2 \times (b_1 + b_2)$.



The required combinator a can be constructed by applying these in parallel and factoring out the results, i.e., $a = (a'_1 \oplus a'_2) \ggg (\texttt{arr}\ \mathit{factor})$



□

5. Arrow Metalanguage over Π^o

The language Π^o extends Π with recursive types and $trace$. We carry over these extensions to get our arrow metalanguage ML_{Π^o} from ML_{Π}. Note that the addition of recursive types of the form $\mu x.b$ and their *fold/unfold* combinators do not affect the arrow structure. However, as we did for other composition combinators, $\fatsemi$ and $\times/+$, the inclusion of composition combinator *trace* requires us to include an effectful variant of the same. This composition combinator is called `traceA`.

Definition 5.1 (Syntax of ML_{Π^o})**.**

$$\begin{array}{rcl}
\textit{base types}, b & = & 1 \mid b \times b \mid b + b \mid \mu x.b \mid x \\
\textit{values}, v & = & () \mid (v, v) \mid \textit{left } v \mid \textit{right } v \mid \langle v \rangle \\
\textit{types}, t & ::= & b \rightleftharpoons b \mid b \rightharpoonup b \\
\textit{arrow comp.}, a & ::= & \textit{iso} \mid a + a \mid a \times a \mid a \fatsemi a \mid \textit{trace } a \\
 & \mid & \texttt{arr } a \mid a \ggg a \mid \texttt{first } a \mid \texttt{left } a \\
 & \mid & \texttt{traceA } a \mid \texttt{create}_b \mid \texttt{erase}
\end{array}$$

The arrow type $\rightharpoonup$ is analogous to $\rightsquigarrow$ of ML_{Π}: we use a different symbol to emphasize that the underlying bijections are partial. The typing rule and semantics for `traceA` are as follows:

Definition 5.2. *Types and Semantics of* `traceA`

Types:

$$\frac{a : b_1 + b_2 \rightharpoonup b_1 + b_3}{\texttt{traceA } a : b_2 \rightharpoonup b_3}$$

Semantics:

$$\frac{(c \ggg \texttt{loop}_a)\ (\textit{right } v_1) \mapsto v_2}{(\texttt{traceA } a)\ v_1 \mapsto v_2} \qquad \frac{(c \ggg \texttt{loop}_a)\ (\textit{left } v_1) \mapsto v_2}{\texttt{loop}_a\ (\textit{left } v_1) \mapsto v_2} \qquad \frac{}{\texttt{loop}_a\ (\textit{right } v) \mapsto v}$$

The semantics of `traceA` is similar to that of *trace* but with an important technical difference. In contrast to *trace* which defines looping computations whose bodies are (partial) bijections, `traceA` can be used to define looping computations whose bodies may create and erase information. The body of the `traceA`-loop erase information repeatedly which `traceA` reflects in its $\rightharpoonup$ type.

Proposition 5.3. *ML_{Π^o} respects the arrow laws*

PROOF. These can be checked as before. □

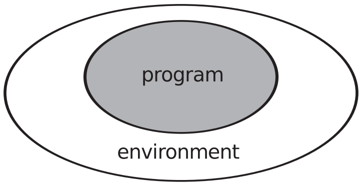
7

Effects as Interactions

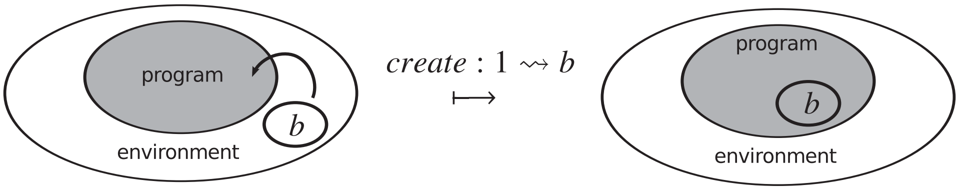
This chapter shows that conventional irreversible computation can be compiled to our information preserving model. This compilation is interesting for two reasons: First, it shows that there are implicit information effects in conventional irreversible computational models and that *these implicit effects can be made explicit*. Second, it shows that information effects *can be compiled away*, i.e. a program with information effects can be compiled into a program that does not have information effects.

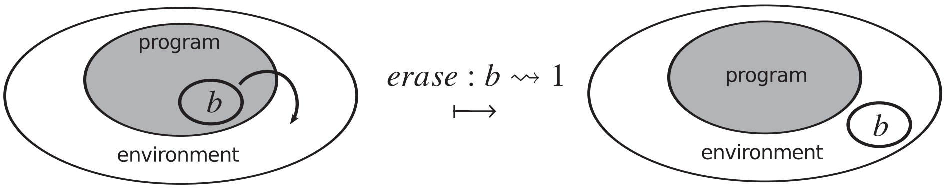
Compiling away an information effects to pure Π/Π^o is similar to how monadic effects (such as the `State` or `Cont` monads in Haskell) can be compiled to a typed λ-calculus that does not actually natively contain those effects. Indeed, not all monads can be compiled away, for example `STM` or `IO` are built into Haskell and cannot be expressed as source level transformations.

When compiled, information effects take the form of interactions with an *information environment* as explained below. One may imagine the information contained by the program as the gray inner bubble in the diagram below. The information environment is an outer layer of information surrounding the computation. The information environment is implicit in conventional computing models and when compiled to Π/Π^o it is made explicit.



The operator `create` : $1 \rightsquigarrow b$ can be though of as exposing information in the environment to the computation and `erase` : $b \rightsquigarrow 1$ can be thought of as moving information from the computation to the environment.





The treatment of information effects has a parallel with open and closed systems in physics. Closed physical systems conserve mass and energy and are the basic unit of study in physics. Open systems interact with their environment, possibly exchanging mass or energy. These interactions may be thought of as *effects* that modify the conservation properties of the system. Computations with information effects are much like open systems and they can be converted into pure computations by making explicit the surrounding information environment with which they interact.

We start by defining a canonical irreversible source language. The compilation process is type-directed and has two phases. First, we compile the source language to our arrow metalanguage. Second, we compile information effects away. This compilation is further partitioned into two: the compilation of the strong normalizing fragment of the source language to Π is presented separately from the compilation of full Turing complete source language to Π^o.

1. Reversible Logic

Toffoli's pioneering work on reversible models of computation [Toffoli, 1980] established the following fundamental theorem.

Theorem 1.1 (Toffoli)**.** *For every finite function $\phi : bool^m \rightarrow bool^n$ there exists an invertible finite function $\phi^R : bool^{r+m} \rightarrow bool^{r+m}$, with $r \leq n$, such that $\phi(x_1, \ldots, x_m) = (y_1, \ldots, y_n)$ iff*

$$\phi^R(x_1, \ldots, x_m, \overbrace{\mathit{false}, \ldots, \mathit{false}}^{r}) = (\overbrace{\cdots}^{m+r-n}, y_1, \ldots, y_n)$$

The proof of the theorem is constructive. Intuitively, the function ϕ is "compiled" to a reversible function ϕ^R which takes extra arguments and produces extra results. When the extra arguments are each fixed to the constant value false and the extra results are ignored, the reversible function behaves exactly like the original function. For example, the function $and : bool^2 \to bool$ can be compiled to the function $toffoli : bool^3 \to bool^3$ which behaves as follows:

$toffoli(v_1, v_2, v_3) =$

$if\ (v_1\ and\ v_2)\ then\ (v_1, v_2, not(v_3))\ else\ (v_1, v_2, v_3)$

A quick examination of the truth table of the $toffoli$ function shows that it is reversible. Moreover, we can check that:

$toffoli(v_1, v_2, false) =$

$if\ (v_1\ and\ v_2)\ then\ (v_1, v_2, true)\ else\ (v_1, v_2, false)$

which confirms that we can recover and if we ignore the first two outputs. Toffoli's fundamental theorem already includes some of the basic ingredients of our results. Specifically, it establishes that:

- it is possible to base the computation of finite functions on reversible functions;
- irreversible functions are special cases of reversible functions which interact with a global heap (which supplies the fixed constant values) and a global garbage dump (which absorbs the undesired results); and
- it is possible to translate irreversible functions to reversible functions to expose the heap and garbage.

In this context, our results can be seen as extending Toffoli's in the following ways:

- Instead of working with truth tables, we work with a rich type structure and use (partial) bijections between the types;
- We introduce term languages for irreversible and reversible computations and develop a type-directed compositional translation;
- We extend the entire framework to deal with infinite functions, e.g., on the natural numbers.
- We establish that the manipulations of the heap and the garbage constitute computational effects that can be tracked by the type system.

2. Source Language

As our canonical irreversible language we use a simply-typed, first-order functional language with sums and pairs and *for* loops. We present this language as two fragments: the first of these, LET, is strongly normalizing, and the second, LET^o, is Turing complete and includes natural numbers and *for* loops for iteration. The language is fairly conventional and is presented without much discussion.

Definition 2.1. *Syntax of LET*

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

The environments, ρ, in the definition above are runtime environments that bind variables to values and are used in the definition of the operational semantics. They must not be confused with type environments, Γ, or with the information environments discussed previously.

Definition 2.2. *Type system of LET*

$$\frac{\Gamma(x) = b}{\Gamma \vdash x : b} \quad \frac{\Gamma \vdash e_1 : b_1 \quad \Gamma, x : b_1 \vdash e_2 : b_2}{\Gamma \vdash \textit{let } x = e_1 \textit{ in } e_2 : b_2}$$

$$\frac{}{\Gamma \vdash () : 1} \quad \frac{\Gamma \vdash e_1 : b_1 \quad \Gamma \vdash e_2 : b_2}{\Gamma \vdash (e_1, e_2) : b_1 \times b_2}$$

$$\frac{\Gamma \vdash e : b_1}{\Gamma \vdash \textit{left } e : b_1 + b_2} \quad \frac{\Gamma \vdash e : b_2}{\Gamma \vdash \textit{right } e : b_1 + b_2}$$

$$\frac{\Gamma \vdash e : b_1 \times b_2}{\Gamma \vdash \textit{fst } e : b_1} \quad \frac{\Gamma \vdash e : b_1 \times b_2}{\Gamma \vdash \textit{snd } e : b_2}$$

$$\frac{\Gamma \vdash e : b_1 + b_2 \quad \Gamma, x : b_1 \vdash e_1 : b \quad \Gamma, x : b_2 \vdash e_2 : b}{\Gamma \vdash \textit{case } e \; x.e_1 \; x.e_2 : b}$$

As a simple example, the not operator over the type $bool = 1 + 1$ can be macro encoded as:

$not\ x = case\ x\ [y.right\ ()]\ [y.left\ ()]$.

The most interesting aspect of LET is that expressions may freely erase and duplicate data in irreversible ways. For instance, $fst\ (v_1, v_2)$ erases the information content of v_2.

Definition 2.3. *Typing rules for environments:*

$$\frac{}{\vdash \epsilon : \epsilon} \quad \frac{\vdash \rho : \Gamma \quad \vdash v : b}{\vdash (\rho, x = v) : (\Gamma, x : b)}$$

The operational semantics of the LET is entirely straightforward. It is presented as '$\mapsto_{let}$' transitions of machines states that are tuples of the form $\langle e, \rho\rangle$ and is presented in a conventional big-step style. We say $eval_{let}(e) = v$ if $\langle e, \epsilon\rangle \mapsto^*_{let} v$.

Definition 2.4. *Operational Semantics for LET*

$$\frac{\rho(x) = v}{\langle x, \rho\rangle \mapsto_{let} v} \quad \frac{\langle e, \rho\rangle \mapsto_{let} v}{\langle left\ e, \rho\rangle \mapsto_{let} left\ v} \quad \frac{\langle e, \rho\rangle \mapsto_{let} v}{\langle right\ e, \rho\rangle \mapsto_{let} right\ v}$$

$$\frac{\langle e_1, \rho\rangle \mapsto_{let} v_1 \quad \langle e_2, \rho; x = v_1\rangle \mapsto_{let} v_2}{\langle let\ x = e_1\ in\ e_2, \rho\rangle \mapsto_{let} v_2}$$

$$\frac{\langle e, \rho\rangle \mapsto_{let} left\ v_1 \quad \langle e_1, \rho; x = v_1\rangle \mapsto_{let} v}{\langle case\ e\ x.e_1\ x.e_2, \rho\rangle \mapsto_{let} v} \quad \frac{\langle e, \rho\rangle \mapsto_{let} right\ v_2 \quad \langle e_2, \rho; x = v_2\rangle \mapsto_{let} v}{\langle case\ e\ x.e_1\ x.e_2, \rho\rangle \mapsto_{let} v}$$

$$\frac{\langle e_1, \rho\rangle \mapsto_{let} v_1 \quad \langle e_2, \rho\rangle \mapsto_{let} v_2}{\langle (e_1, e_2), \rho\rangle \mapsto_{let} (v_1, v_2)} \quad \frac{\langle e, \rho\rangle \mapsto_{let} (v_1, v_2)}{\langle fst\ e, \rho\rangle \mapsto_{let} v_1} \quad \frac{\langle e, \rho\rangle \mapsto_{let} (v_1, v_2)}{\langle snd\ e, \rho\rangle \mapsto_{let} v_2}$$

The extended language, LET^o, includes additionally nats (numbers), operations on nats and a for loop:

Definition 2.5. *Syntax of LET^o: LET^o extends LET with the following syntactic forms:*

$$\begin{array}{rcl} Base\ types, b & = & ... \mid nat \\ Values, v & = & ... \mid n \\ Expressions, e & = & ... \mid n \mid add_1\ e \mid sub_1\ e \mid iszero?\ e \\ & \mid & for(x = e_1; e_2; e_3) \end{array}$$

Definition 2.6. *Type System of LET^o: LET^o extends the type judgments of LET with the following derivations:*

$$\frac{}{\Gamma \vdash n : nat} \quad \frac{\Gamma \vdash e : nat}{\Gamma \vdash add_1\ e : nat} \quad \frac{\Gamma \vdash e : nat}{\Gamma \vdash sub_1\ e : nat} \quad \frac{\Gamma \vdash e : nat}{\Gamma \vdash iszero?\ e : bool}$$
$$\frac{\Gamma \vdash e_1 : b \quad \Gamma, x : b \vdash e_2 : bool \quad \Gamma, x : b \vdash e_3 : b}{\Gamma \vdash for(x = e_1; e_2; e_3) : b}$$

The for loop above first evaluates e_1 and binds the result v to x in the environment of e_2. If the evaluation of e_2 returns $false$, the evaluation of for return v as the result. If the e_2 returns $true$, e_3 is evaluated in the environment where x is bound to v and whose result v' becomes the new values of x.

The most interesting aspect of the extended language LET^o is that it admits partial functions. Here is an example of iterative addition of two numbers n and m in this syntax:

$snd\ (for\ x = (n, m); not\ (iszero?\ (fst\ x)); (sub_1\ (fst\ x), add_1\ (snd\ x)))$

Definition 2.7. *Operational Semantics of LET^o*

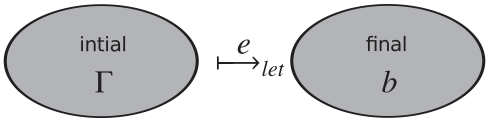
$$\frac{}{\langle n, \rho\rangle \mapsto_{let} n} \quad \frac{\langle e, \rho\rangle \mapsto_{let} n}{\langle add_1\ e, \rho\rangle \mapsto_{let} n+1} \quad \frac{\langle e, \rho\rangle \mapsto_{let} n+1}{\langle sub_1\ e, \rho\rangle \mapsto_{let} n}$$
$$\frac{\langle e, \rho\rangle \mapsto_{let} 0}{\langle iszero?\ e, \rho\rangle \mapsto_{let} true} \quad \frac{\langle e, \rho\rangle \mapsto_{let} 0}{\langle iszero?\ e, \rho\rangle \mapsto_{let} false}$$
$$\frac{\begin{array}{c}\langle e_1, \rho\rangle \mapsto_{let} v \\ \langle e_2, \rho; x = v\rangle \mapsto_{let} false\end{array}}{\langle for(x = e_1; e_2; e_3), \rho\rangle \mapsto_{let} v} \quad \frac{\begin{array}{c}\langle e_1, \rho\rangle \mapsto_{let} v \quad \langle e_2, \rho; x = v\rangle \mapsto_{let} true \\ \langle for(x = e_3; e_2; e_3), \rho; x = v\rangle \mapsto_{let} v'\end{array}}{\langle for(x = e_1; e_2; e_3), \rho\rangle \mapsto_{let} v'}$$

Note that in the above definition we use $n+1$ to denote the numerical value of adding the 1 to n. Further, $true$, $false$ and $bool$ are syntactic sugar for $left\ ()$, $right()$ and the type $1+1$.

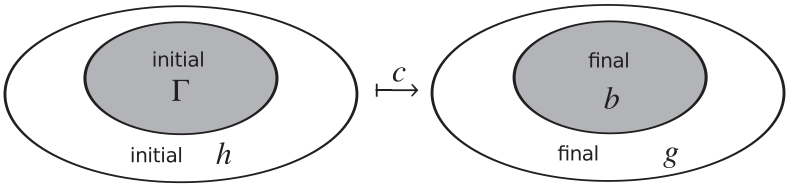
3. Metalanguages and Translations: An Overview

The technical goal of the remaining sections of this chapter is to translate the source language LET^o with irreversible primitives to the target information-preserving language Π^o. Computations in LET^o have the general form $\Gamma \vdash e : b$. Abstractly, the execution of a

LET^o computation can be viewed as a map from the initial information state of the program to the final information state of the program.



The compilation of $\Gamma \vdash e : b$ produces a Π^o combinator that not only maps the inputs of e to the outputs of e, i.e. maps Γ to b, but also makes explicit its interactions with an information environment. We give the initial environment the type h and the final environment the type g. Thus target of compilation has the form $c : h \times \Gamma \rightleftharpoons g \times b$ and whose evaluation may be abstractly pictured as:



As explained before, this translation is broken up into two phases.

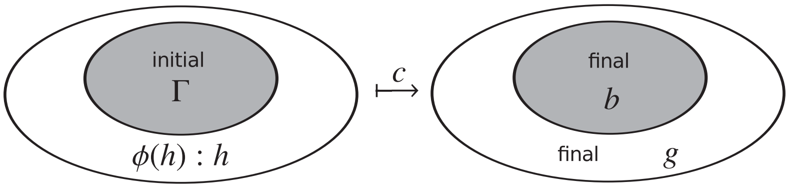
3.1. Translation from Source to Metalanguage. In the first phase we translate from the source language to the metalanguage of information effects. The translation from the source language to the metalanguage essentially exposes the implicit erasure and duplication of environment bindings.

For example, the evaluation rule of a pair (see Sec. 2) duplicates the environment which can only be done in the metalanguage using explicit occurrences of the `create` effect combinator. Similarly, the evaluation of a variable projects one value out of the environment, implicitly erasing the rest of the environment, which again can only be done using explicit occurrences of the `erase` effect combinator.

This phase of the translation converts LET^o programs of the form $\Gamma \vdash e : b$ to ML_{Π^o} combinators of the form $a : \Gamma^\times \rightsquigarrow b$. Here $\Gamma^\times$ is an encoding of the type environment Γ. The resulting combinator a has explicit `create` and `erase` operations. Technically, we define two translations $\mathcal{T}_1 : \mathsf{LET} \Longrightarrow \mathsf{ML}_\Pi$ and $\mathcal{T}_1^o : \mathsf{LET}^o \Longrightarrow \mathsf{ML}_{\Pi^o}$. The first maps the strongly-normalizing subset of the source to ML_Π (Sec. 4) and the second is an extension that handles the full source and targets ML_{Π^o} (Sec. 6.1). For $\mathcal{T}_1^o$ we don't deal with the full language ML_{Π^o}, but only the fragment that is the compilation target. This also simplifies the subsequent translation, $\mathcal{T}_2^o$, as we will see.

3.2. Translation from Metalanguage to Target. This translation needs to compile the effect combinators to the target language. Effects are viewed as interaction with an information environment, which is made explicit in the type of the compilation target. The basic scheme is based on Toffoli's idea described in Sec. 1: an irreversible function of type $a \to b$ is translated to a bijection $(h, a) \leftrightarrow (g, b)$ where h is the type of the heap that supplies the constant values and g is the type of the garbage that absorbs the un-interesting and un-observable outputs. In our setting, the translation takes ML_{Π} combinators of the form $a : \Gamma^{\times} \rightsquigarrow b$ and produces Π combinators of the form $c : h \times \Gamma^{\times} \leftrightarrow g \times b$. Once the primitive effect combinators, `create` and `erase`, have been translated, the translation of the arrow combinators thread the heap and garbage through more complex computations.

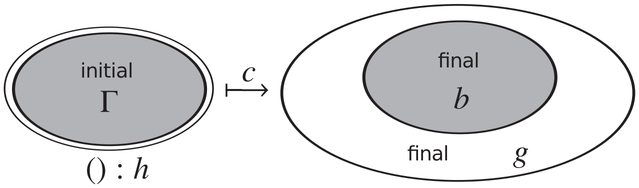
Technically, we again have two translations: one for the strongly-normalizing subset of the source language and one for the full source language. The first translation $\mathcal{T}_2 :: \mathsf{ML}_{\Pi} \Longrightarrow \Pi$ (Sec. 5) selects particular values for the heap and garbage to embed the irreversible effects into bijections. [1]



The second translation $\mathcal{T}_2^o :: \mathsf{ML}_{\Pi^o} \Longrightarrow \Pi^o$ (Sec. 7) works for the full language. It has an important difference which arises from the fact that there is an inherent asymmetry between `create` and `erase`: the operator `create` is always used to create a known constant while `erase` is used to erase information that is only known at run time. This inherent

[1] This idea has been the basis of translations similar to ours [Altenkirch and Grattage, 2005]. The literature also includes translations for other languages [Abramsky, 2005b, Di Pierro et al., 2006, Glück and Kawabe, 2005, Huelsbergen, 1996, Kluge, 2000, Ross, 1997, Zuliani, 2001] that share some of the intuition of the translation we present but differ significantly in the technical details.

asymmetry of the operators is a consequence of the fact that the source language is (forward) deterministic, but lacks backward determinism. As we saw in Sec. 3, Π^o can express the creation and erasure of constants and we leverage this to completely eliminate the heap ($h = 1$) in favor of using `traceA`.



Thus the later translation takes ML_{Π^o} combinators of the form $a : \Gamma^\times \rightsquigarrow b$ to Π^o combinators of the form $c : 1 \times \Gamma^\times \rightleftharpoons g \times b$. If LET^o had an operation that introduced values unknown at compile time, such as an input operation or a random number generator, we would have to re-introduce the heap.

4. Translation from LET to ML$_\Pi$

The translation $\mathcal{T}_1$ maps a closed term of type b in LET to an ML$_\Pi$ combinator $c : 1 \rightsquigarrow b$. As the translation is type-directed, it must also handle terms with free variables that are supplied by an environment. We discuss the encoding of type environments, runtime environments and the operation that looks up the value of a variable in the environment, all of which are used by the translation.

4.1. Environments. A LET type environment Γ is translated to an ML$_\Pi$ type as follows:

$[\epsilon]^\times = 1$

$[\Gamma, x : b]^\times = [\Gamma]^\times \times b$

A runtime environment $\rho : \Gamma$ is translated to a value $v_\rho : \Gamma^\times$:

$[\epsilon]^\times = ()$

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

Lemma 4.1 (Lookup). *If $\Gamma \vdash x : b$ and $\Gamma^\times$ is the encoding of Γ, then there exists a combinator $a_{lookup(x)} : \Gamma^\times \rightsquigarrow b$ that looks up x in $\Gamma^\times$.*

PROOF. The required combinator a depends on the structure of Γ:

- Case ϵ: This cannot arise because Γ must contain x.
- Case $\Gamma', x' : b'$ and $x' = x$: Then $a = \texttt{sndA}$
- Case $\Gamma', x' : b'$ and $x' \neq x$: Then we know that the required x must be bound in Γ', i.e. $\Gamma' \vdash x : b$. Thus by induction there exists $a' : [\Gamma']^{\times} \rightsquigarrow b$. So the required combinator is $a = (a' \otimes id) \ggg \texttt{fstA}$.

□

4.2. The Translation $\mathcal{T}_1$. We translate a LET judgment of the form $\Gamma \vdash e : b$ to an ML$_{\Pi}$ combinator $a : \Gamma^{\times} \rightsquigarrow b$ in such a way that the execution of the resulting ML$_{\Pi}$ term simulates the execution of the original term. Because the evaluation of LET expressions requires an environment ρ, the evaluation of the translated combinator must be given a value v_{ρ} of type $\Gamma^{\times}$ denoting the value of the environment.

Lemma 4.2 ($\mathcal{T}_1$ and its correctness)**.** *For any well typed LET expression $\Gamma \vdash e : b$, $\mathcal{T}_1[\Gamma \vdash e : b]$ gives us a combinator 'a' and a type $\Gamma^{\times}$ in ML$_{\Pi}$ such that:*

(1) $a : \Gamma^{\times} \rightsquigarrow b$

(2) $\forall\ (\rho : \Gamma), \exists\ (v : b).$ *if* $\langle e, \rho \rangle \mapsto^{*}_{let} v$ *then* $a\ [\rho]^{\times} \mapsto^{*}_{ML} v.$

The proof is structured as follows. For each type-derivation in LET of the form

$$\frac{\Gamma_1 \vdash e_1 : b_1}{\Gamma \vdash e : b}$$

we will write the expected output as

$$\frac{\Gamma_1 \vdash e_1 : b_1 \dashrightarrow a_1 : \Gamma_1^{\times} \rightsquigarrow b_1}{\Gamma \vdash e : b \dashrightarrow a : \Gamma^{\times} \rightsquigarrow b}$$

This is to be read follows: for the derivation of $\Gamma \vdash e : b$ we have to produce an $a : \Gamma^{\times} \rightsquigarrow b$ such that the evaluation of a respects the evaluation of e. Further, for each antecedent of the form $\Gamma_1 \vdash e_1 : b_1$ by induction we have $a_1 : \Gamma_1^{\times} \rightsquigarrow b_1$ that respects its evaluation. We simultaneously present the type-directed translation and prove its correctness.

- Case $()$:

$$\frac{}{\Gamma \vdash () : 1 \dashrightarrow \texttt{erase} : \Gamma^{\times} \rightsquigarrow 1}$$

We have that `erase` $v_\rho \mapsto_{ML} ()$.

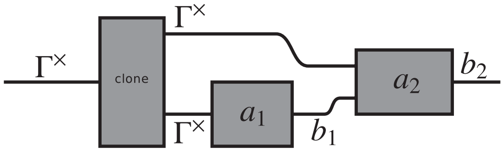
- Case x:

$$\frac{\Gamma(x) = b}{\Gamma \vdash x : b \dashrightarrow a_{lookup(x)} : \Gamma^\times \rightsquigarrow b}$$

- Case $let\ x = e_1\ in\ e_2$:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : b_1 \dashrightarrow a_1 : \Gamma^\times \rightsquigarrow b_1 \\ \Gamma, x : b_1 \vdash e_2 : b_2 \dashrightarrow a_2 : \Gamma^\times \times b_1 \rightsquigarrow b_2\end{array}}{\Gamma \vdash let\ x = e_1\ in\ e_2 : b_2 \dashrightarrow a : \Gamma^\times \rightsquigarrow b_2}$$

To construct the required a we first clone $\Gamma^\times$. We can apply a_1 to one of the copies to get b_1. The resulting value of type $\Gamma^\times \times b_1$ is the input required by a_2 which returns the result of type b_2:



Thus the required combinator is:

$a = clone_{\Gamma^\times} \ggg (\texttt{second}\ a_1) \ggg a_2$

- Case (e_1, e_2):

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : b_1 \dashrightarrow a_1 : \Gamma^\times \rightsquigarrow b_1 \\ \Gamma \vdash e_2 : b_2 \dashrightarrow a_2 : \Gamma^\times \rightsquigarrow b_2\end{array}}{\Gamma \vdash (e_1, e_2) : b_1 \times b_2 \dashrightarrow a : \Gamma^\times \rightsquigarrow b_1 \times b_2}$$

As with *let*, we clone $\Gamma^\times$ and use each copy to create one component of the pair. Thus we have:

$a = clone_{\Gamma^\times} \ggg (a_1 \otimes a_2)$

- Cases *fst* e, *snd* e:

$$\frac{\Gamma \vdash e : b_1 \times b_2 \dashrightarrow a_1 : \Gamma^\times \rightsquigarrow b_1 \times b_2}{\Gamma \vdash fst\ e : b_1 \dashrightarrow a : \Gamma^\times \rightsquigarrow b_1}$$

We have $a = a_1 \ggg$ `fstA`. And similarly for $snd\ e$ we have $a = a_1 \ggg$ `sndA`.

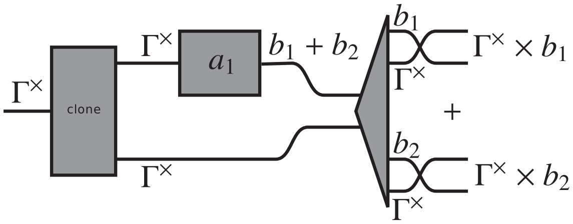
- Cases *left* e, *right* e:

$$\frac{\Gamma \vdash e : b_1 \dashrightarrow a_1 : \Gamma^\times \rightsquigarrow b_1}{\Gamma \vdash left\ e : b_1 + b_2 \dashrightarrow a : \Gamma^\times \rightsquigarrow b_1 + b_2}$$

We have $a = a_1 \ggg$ `leftA` and similarly for *right* e we have $a = a_1 \ggg$ `rightA`.

- Case *case e x.e*$_1$ *x.e*$_2$:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : b_1 + b_2 \dashrightarrow a_1 : \Gamma^\times \rightsquigarrow b_1 + b_2 \\ \Gamma, x : b_1 \vdash e_2 : b_3 \dashrightarrow a_2 : \Gamma^\times \times b_1 \rightsquigarrow b_3 \\ \Gamma, x : b_2 \vdash e_3 : b_3 \dashrightarrow a_3 : \Gamma^\times \times b_2 \rightsquigarrow b_3\end{array}}{\Gamma \vdash \mathit{case}\ e_1\ x.e_2\ x.e_3 : b_3 \dashrightarrow a : \Gamma^\times \rightsquigarrow b_3}$$



Here we have cloned $\Gamma^\times$ and constructed $b_1 + b_2$ using one copy of $\Gamma^\times$ and a_1. We then distributed $\Gamma^\times$ over $b_1 + b_2$ and resulting in two possible environments $\Gamma^\times \times b_1$ or $\Gamma^\times \times b_2$. At this point, we can apply a_2 and a_3 to these environments resulting in $b_3 + b_3$ which we can *join* to get the desired result b_3. Thus we have:

$$a = \mathit{clone}_{\Gamma^\times} \ggg (\texttt{left}\ c_1) \ggg (\texttt{arr}\ \mathit{distrib}) \ggg$$
$$((\texttt{arr}\ \mathit{swap}^\times) \oplus (\texttt{arr}\ \mathit{swap}^\times)) \ggg (a_2 \oplus a_3) \ggg \mathit{join}$$

5. Translation from ML$_\Pi$ to Π

The translation $\mathcal{T}_2$ maps an ML$_\Pi$ combinator $a : b_1 \rightsquigarrow b_2$ to an isomorphism $h \times b_1 \leftrightarrow g \times b_2$. The types h and g are determined based on the structure of the combinator a and are fixed by the translation $\mathcal{T}_2$. The translation is set up such that when we supply $\phi(h)$ for the heap along with the given input value of type b_1, the compiled combinator produces some unspecified value for g and the value for b_2 that the original arrow combinator would have produced.

Lemma 5.1 ($\mathcal{T}_2$ and its correctness)**.** *For any ML*$_\Pi$ *combinator* $a : b_1 \rightsquigarrow b_2$*,* $\mathcal{T}_2[a : b_1 \rightsquigarrow b_2]$ *gives us* c*,* h *and* g *in* Π *such that:*

- $c : h \times b_1 \leftrightarrow g \times b_2$
- $\forall\ (v_1 : b_1), \exists\ (v_g : g), (v_2 : b_2)$ *if* $a\ v_1 \mapsto^*_{ML} v_2$ *then* $c\ (\phi(h), v_1) \mapsto^* (v_g, v_2)$.

As before, we present the translation along with the proof of correctness.

- `arr` a:

$$\frac{a : b_1 \leftrightarrow b_2}{\texttt{arr}\ a : b_1 \rightsquigarrow b_2 \dashrightarrow c : h \times b_1 \leftrightarrow g \times b_2}$$

To construct the required c we choose $h = g = 1$ and thus we have $c = id \times a$. It is easy to verify that $c\ ((), v_1) \mapsto ((), v_2)$ assuming $a\ v_1 \mapsto v_2$.

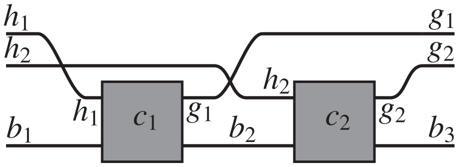
- $a_1 \ggg a_2$:

$$\frac{\begin{array}{c} a_1 : b_1 \rightsquigarrow b_2 \dashrightarrow c_1 : h_1 \times b_1 \leftrightarrow g_1 \times b_2 \\ a_2 : b_2 \rightsquigarrow b_3 \dashrightarrow c_2 : h_2 \times b_2 \leftrightarrow g_2 \times b_3 \end{array}}{a_1 \ggg a_2 : b_1 \rightsquigarrow b_3 \dashrightarrow c : h \times b_1 \leftrightarrow g \times b_3}$$

We choose $h = h_1 \times h_2$ and $g = g_1 \times g_2$ and we have

$c = \ assocr^{\times} \fatsemi (id \times c_1) \fatsemi \mathit{shuffle} \fatsemi (id \times c_2) \fatsemi assocl^{\times}$

where $\mathit{shuffle} = (assocl^{\times} \fatsemi (swap^{\times} \times id) \fatsemi assocr^{\times})$.



- `first` a: Given that $a : b_1 \rightsquigarrow b_2$ translates to $c_1 : h_1 \times b_1 \leftrightarrow g_1 \times b_2$, we translate `first` $a : b_1 \times b_3 \rightsquigarrow b_2 \times b_3$ to $c : h \times (b_1 \times b_3) \leftrightarrow g \times (b_2 \times b_3)$ where $h = h_1$, $g = g_1$, and $c = assocl^{\times} \fatsemi\ (c_1 \times id)\ \fatsemi assocr^{\times}$

- `left` a: Given that $a : b_1 \rightsquigarrow b_2$ translates to $c_1 : h_1 \times b_1 \leftrightarrow g_1 \times b_2$, we translate `left` $a : b_1 + b_3 \rightsquigarrow b_2 + b_3$ to $c : h \times (b_1 + b_3) \leftrightarrow g \times (b_2 + b_3)$ as explained next. Assume $c_1\ (\phi(h_1), v_1) \mapsto (v_{g_1}, v_2)$, we need to prove:

$c\ (\phi(h), \mathit{left}\ v_1) \mapsto (v_{g'}, \mathit{left}\ v_2)$

$c\ (\phi(h), \mathit{right}\ v_3) \mapsto (v_{g''}, \mathit{right}\ v_3)$

Here $v_{g'}$ and $v_{g''}$ have the type g and $v_1 : b_1$, $v_2 : b_2$ and $v_3 : b_3$. We define the required types h and g as shown below.

$h = (h_1 \times ((b_2 + b_3) \times (b_3 + b_2)))$

$g = g' + g''$

where:

$g' = (g_1 \times (b_2 \times (b_3 + b_2)))$

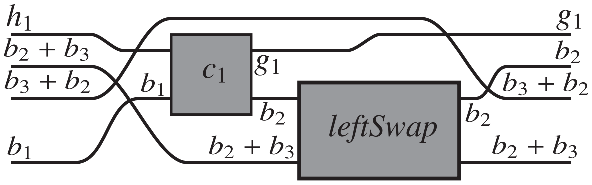
$g'' = (h_1 \times ((b_2 + b_3) \times b_3))$

We construct the required c in terms of two combinators c' and c'' that handle the *left* and *right* cases of the input $b_1 + b_2$. We first construct the combinator c' that has the following type and semantics:

$c' : b_1 \times h \leftrightarrow g' \times (b_2 + b_3)$

$c'(v_1, \phi(h)) \mapsto (v_{g_1}, \mathit{left}\ v_2)$

To construct c', we need the *leftSwap* combinator that was defined in Sec. 4.2. We can draw a wiring diagram for c' as follows:

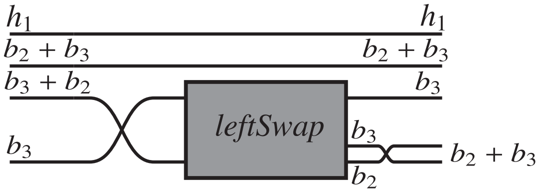


The combinator c' uses c_1 to obtain a value of type b_2 and then uses *leftSwap* to construct a value of type $b_2 + b_3$. We can now define c'' that works on the *right* branch to be:

$c'' : b_3 \times h \leftrightarrow g'' \times (b_2 + b_3)$

$c''(v_3, \phi(h)) \mapsto (v_{g_2}, \mathit{right}\ v_3)$

As before we need the *leftSwap* combinator, but this time at the type $b_3 \times (b_3 + b_2)$. The wiring diagram for c'' is shown:



The definition of c'' takes the value of type b_3 and constructs a value of type $b_2 + b_3$ using *leftSwap* and *swap*$^+$. Given the construction of c' and c'' we can construct the required c as:

$c : h \times (b_1 + b_3) \leftrightarrow g \times (b_2 + b_3)$

$c = \mathit{swap}^\times \fatsemi \mathit{distrib} \fatsemi ((\mathit{swap}^\times \fatsemi c') \times (\mathit{swap}^\times \fatsemi c'')) \fatsemi \mathit{factor}$

- `create`:

$$\frac{}{\texttt{create} : 1 \rightsquigarrow b \dashrightarrow c : h \times 1 \leftrightarrow g \times b}$$

We choose $h = b$ and $g = 1$ and we have $c = swap^{\times}$. The definition of `create` is simple because we have taken care to correctly thread a value of type h and `create` simply reifies this value.

- `erase`:

$$\frac{}{\texttt{erase} : b \rightsquigarrow 1 \ \dashrightarrow \ e : h \times b \leftrightarrow g \times 1}$$

We choose $h = 1$ and $g = b$ and we have $c = swap^{\times}$. Note that this is operationally the same as `create`. The difference is in the types. Since we have set up the rest of the computation to thread the value of type g through and never expose it, to erase a value we simply have to move it to the garbage.

6. Translation from LET^o to $\mathsf{ML}_{\Pi^o}^{nat}$

To keep the compilation simple, we focus on only the fragment of ML_{Π^o} terms that are produced by the compilation from LET^o. This fragment is ML_{Π} extended with loops (via `traceA`) and only one specific recurive type, namely nat, instead of full recursive types of the form $\mu x.b$. A richer source language with full algebraic data types would have produced the full ML_{Π^o} as the target.

Definition 6.1 (Syntax of $\mathsf{ML}_{\Pi^o}^{nat}$).

$$\begin{array}{rcl}
base\ types, b & = & 1 \mid b \times b \mid b + b \mid nat \\
values, v & = & () \mid (v, v) \mid left\ v \mid right\ v \mid n \\
types, t & ::= & b \rightleftharpoons b \mid b \rightharpoonup b \\
arrow\ comp., a & ::= & iso \mid a + a \mid a \times a \mid a \fatsemi a \mid trace\ a \\
 & \mid & \texttt{arr}\ a \mid a \ggg a \mid \texttt{first}\ a \mid \texttt{left}\ a \\
 & \mid & \texttt{traceA}\ a \mid \texttt{create}_b \mid \texttt{erase}
\end{array}$$

The types extend the finite types with nat which is an abbreviation for $\mu x.1 + x$. In addition to the usual finite values, we also include natural numbers n which are abbreviations for sequences of *right*-applications that end with *left* $()$. We define $\phi(nat) = 0$ and hence $\texttt{create}_{nat} = 0$. The set of underlying isomorphisms extends the ones for finite types with $unfold : nat \rightleftharpoons 1 + nat : fold$.

Lemma 6.2 (Cloning)**.** *For any type b, we can construct an operator $clone$ of type $b \rightharpoonup b \times b$ such that: $clone\ v \mapsto_{ML} (v, v)$*

PROOF. This is an extension of Lemma 4.1. The only new datatype is nat. We `create` a 0 and use the loop of Sec. 3 to iteratively add_1. □

The required translation $\mathcal{T}^o$: $\mathsf{LET}^o \Longrightarrow \Pi^o$ is factored into $\mathcal{T}_1^o$: $\mathsf{LET}^o \Longrightarrow \mathsf{ML}_{\Pi^o}^{nat}$ and $\mathcal{T}_2^o$: $\mathsf{ML}_{\Pi^o}^{nat} \Longrightarrow \Pi^o$ where the intermediate language ML_{Π^o} extends ML_{Π}.

6.1. Translation $\mathcal{T}_1^o$ from LET^o to $\mathsf{ML}_{\Pi^o}^{nat}$. The translation $\mathcal{T}_1^o$ extends the translation $\mathcal{T}_1$ from LET to ML_{Π}.

Lemma 6.3 ($\mathcal{T}_1^o$ and its correctness)**.** *For any well typed LET^o expression $\Gamma \vdash e : b$, $\mathcal{T}_1^o[\Gamma \vdash e : b]$ gives us a combinator 'a' and a type $\Gamma^\times$ in $\mathsf{ML}_{\Pi^o}^{nat}$ such that:*

(1) $a : \Gamma^\times \rightharpoonup b$
(2) $\forall\ (\rho : \Gamma), \exists\ (v : b)$. *if* $\langle e, \rho \rangle \mapsto_{let}^* v$ *then* $a\ [\rho]^\times \mapsto_{ML}^* v$.

- Cases $add_1\ e$, $sub_1\ e$: These follow from simply lifting the add_1 construction and its adjoint from Sec. 3.

$$\frac{\Gamma \vdash e : nat \dashrightarrow a : \Gamma^\times \rightharpoonup nat}{\Gamma \vdash add_1\ e : nat \dashrightarrow a \fatsemi (\texttt{arr}\ add_1) : \Gamma^\times \rightharpoonup nat}$$

- Case iszero? e:

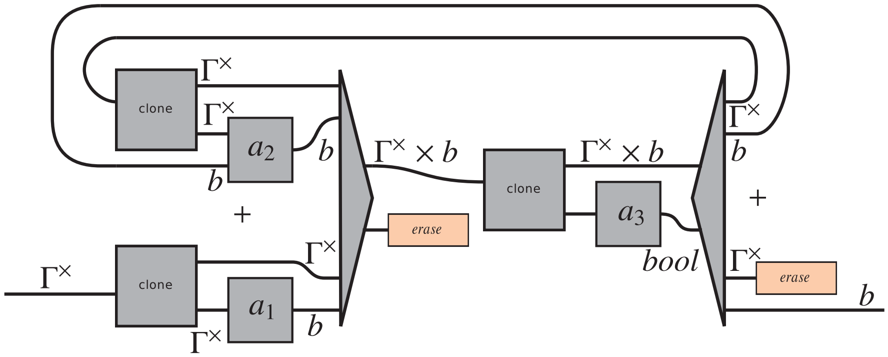
$$\frac{\Gamma \vdash e : nat \dashrightarrow a_1 : \Gamma^\times \rightharpoonup nat}{\Gamma \vdash iszero?\ e : nat \dashrightarrow a : \Gamma^\times \rightharpoonup bool}$$

where $a = a_1 \fatsemi \texttt{arr}\ unfold \fatsemi \texttt{first erase} \fatsemi \texttt{arr}\ unite$

- Case n: The required combinator of type $\Gamma \rightharpoonup nat$ is given by $\texttt{erase} \fatsemi \texttt{create}_{nat} \fatsemi add_1^n$ (where add_1^n is n iterations of add_1).
- Case $for(x = e_1; e_2; e_3)$:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : b \dashrightarrow a_1 : \Gamma^\times \rightharpoonup b \\ \Gamma, x : b \vdash e_2 : bool \dashrightarrow a_2 : \Gamma^\times \times b \rightharpoonup bool \\ \Gamma, x : b \vdash e_3 : b \dashrightarrow a_3 : \Gamma^\times \times b \rightharpoonup b\end{array}}{\Gamma \vdash for(x = e_1; e_2; e_3) : b \dashrightarrow a : \Gamma^\times \rightharpoonup b}$$

The construction is illustrated in the diagram below, in which wires of type 1 used for `erase`, *distrib* etc. have been dropped for the sake of clarity.



Conceptually, each iteration of the `traceA` is determined by the result of a_2. If the conditional is true then the iteration causes a_3 to be executed.

7. Translation from $\mathbf{ML}^{nat}_{\Pi^o}$ to Π^o

Translation $\mathcal{T}_2^o$ is similar to $\mathcal{T}_2$. As discussed in the overview (Sec. 3) the significant difference comes from the fact that Π^o can create constants hence eliminating the need for an input heap type. The translation only needs to track the garbage produced by combinators.

Proposition 7.1. *For any type b of ML$^{nat}_{\Pi^o}$ we can construct $createConst_b : 1 \rightleftharpoons b$ such that $createConst_b\ () \mapsto_{ML} \phi_b$.*

Lemma 7.2 ($\mathcal{T}_2^o$ and its correctness))**.** *For any ML$^{nat}_{\Pi^o}$ combinator $a : b_1 \rightharpoonup b_2$, $\mathcal{T}_2^o[a : b_1 \rightharpoonup b_2]$ gives us c and g in Π^o such that:*

- $c : b_1 \rightleftharpoons g \times b_2$
- $\forall\ (v_1 : b_1), \exists\ (v_g : g), (v_2 : b_2)$. *if* $a\ v \mapsto^*_{ML} v_2$ *then* $c\ v_1 \mapsto^* (v_g, v_2)$.

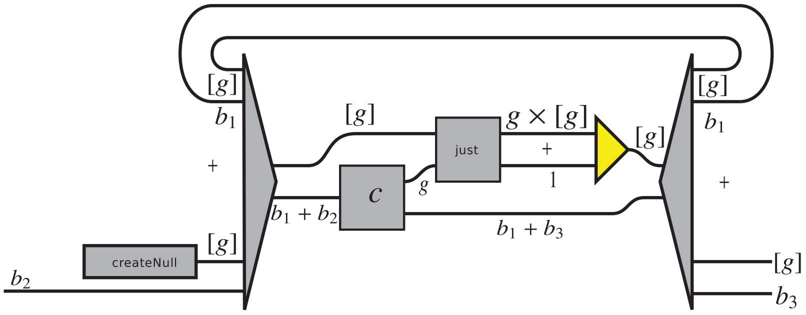
The interesting cases to consider are:

- Case `arr` c: The required combinator is $c \fatsemi uniti : b_1 \rightleftharpoons 1 \times b_2$ where the garbage is 1.
- Case `create`$_b$: The required combinator is $createConst_b \fatsemi uniti : 1 \rightleftharpoons 1 \times b$ with $g = 1$.
- Case `erase`: The required combinator is $uniti \fatsemi swap^\times : 1 \rightleftharpoons b \times 1$ with $g = b$.

- Case `traceA` a:

$$\frac{a : b_1 + b_2 \rightharpoonup b_1 + b_3 \dashrightarrow c : b_1 + b_2 \rightleftharpoons g \times (b_1 + b_3)}{\texttt{traceA}\ a : b_2 \rightharpoonup b_3 \dashrightarrow c_1 : b_2 \rightleftharpoons g' \times b_3}$$

As shown in Sec. 3, we can create and manipulate empty lists of any given type in Π^o. The diagram below is the required combinator c_1 with $g' = [g]$, i.e., the resulting garbage is the list of garbage values produced at each step in the iteration, of type g.



This completes our translations.

7.1. Optimization and Equivalence Preservation.

We have made no attempt to optimize the types h and g generated by the translation to the reversible target language. The fact that the heap and garbage types are not as minimal as possible is akin to space blowup in an unoptimized program. While the sizes of the heap and garbage do not matter in the light of discussing semantics, any direct implementation of this translation may choose to optimize them. Space optimizations as discussed in Toffoli's original paper and further developed by others[Axelsen, 2011] could be incorporated in our translation to minimize the space used for the heap and garbage.

Equivalence Preservation. Equivalent source terms may be translated to terms of *different types*, as the types h and g are chosen by the translation based on the *syntax* of the input terms. In other words, terms that are equal according to source language equalities may not be translated to terms that are equal under target language equalities. In particular, these term vary in the heap and garbage types exposed – i.e. source terms that are equal under the ource language semantics, $\mathsf{LET}/\mathsf{LET}^{o}$, might differ in the information effects they contain and these differences are explicit in the target.

If one desires equalities to be reflected in the target language then some type abstraction will be required over the heap and garbage types. The situation is similar to the closure conversion translation of a compiler which exposes the type of the environment and the fix should follow the same general strategy [Ahmed and Blume, 2008, Minamide et al., 1996].

8
Computing with Rationals

This chapter introduces the notion of duality in the context of Π. At the time of this writing, this is unpublished work and represents an exciting current line of research.

The types of Π are isomorphic to whole numbers – i.e. we have the types 0, 1, 2, 3, etc. The two monoidal structures of Π gives the objects a commutative semiring structure. Unlike traditional studies of the duality of computation [Curien and Herbelin, 2000, Filinski, 1989, Wadler, 2005] which talk about one dualizing operation, in Π we have two axes of duality – we refer to this as an additive duality which is characterized by negative types and a multiplicative duality which is characterized by multiplicative types. Thus including these into Π gives us types that correspond to negative numbers and fractionals. The types thus have a field-like structure. We call this resultant language $\Pi^{\eta\epsilon}$.

Usually duality is discussed in the context of continuations [Curien and Herbelin, 2000, Filinski, 1989, Wadler, 2005] and how they relate to functions and evaluation order. The two dualities discussed here should not be confused with the up/down dualities in settings like polarized focalized linear logic. In $\Pi^{\eta\epsilon}$ we have two function spaces resulting from each notion of duality. In the presence of negative types, the $trace$ operator over the additive monoid can be derived. This $trace$ that is already familiar to us, gives us looping/iteration. Further, we get a new form of $trace$ corresponding to the multiplicative monoid. The multiplicative $trace$ has the ability to search a constraint space which we leverage to construct a SAT solver.

Negative types correspond to the backward flow of information i.e. if one views the left to right direction of the circuits as the positive direction of flow of particles, negative values flow from right to left. Fractional types corresponds to constraints or experiments or measurements. A fractional value of type $1/b$ checks to see if a value of type b has a compatible structure. In the process of checking, both values are consumed. In this sense, fractional types are analogous to measurement vectors in quantum physics.

It is as yet unclear why $\Pi^{\eta\epsilon}$ has two distinct axes of duality, while conventional computation has only one. We suspect that the deep reason underlying this comes from information effects. In the presence of information effects, the category Π with the commutative semiring structure on objects becomes a category with finitary sums and products. In contrast, traditional computation such as LET, the dual calculus of Wadler [2005] or the symmetric lambda calculus of Filinski [1989] has sums and products. For more discussion, see Chapter 9 Section 3.

1. Negative and Fractional Types

The technical goal of this chapter is to extend Π with two axes of duality – negatives and fractionals – thereby extending the type system of the resulting language to correspond to the rational numbers. The natural structure for such a system is the *field of rational numbers*. In other words the natural extension to the semiring structure of Π would be to turn it into a full field.

However, a subtle issue with fields – which goes back to the heart of our motivation of deriving computation from equalities – is that fields have no equational specification. This primarily manifests as the fact that division by the 0 element in a field is left unspecified. We quote Bergstra et al. [2009]:

> The field axioms consist of the equations that define commutative rings and, in particular, two axioms that are not equations that define the inverse operator and the distinctness of the two constants. Now, division is a partial operation, because it is undefined at 0, and the class of fields cannot be defined by any set of equations. Thus, the theory of equational specifications of data types cannot build on the theory of fields ...

Since our motivation is to derive the computational content from equality, it is unclear how we can define a computational model over the rationals. Is there an alternate algebraic structure with an equational specification of division that captures the essence of the rationals? Such a structure exists and is called a 'zero-totalized field' or a *meadow* [Bergstra et al., 2008, 2009]. Meadows are different from fields in the following way:

(1) The meadow axioms do not include the following constraints required in the definition of fields:
   (a) $1/x \times x = 1$ (when $x \neq 0$) (generalized inverse law)
   (b) $0 \neq 1$ (separation law)

(2) and instead include the following two axioms which are applicable for all x:

  (a) $x \times (x \times (1/x)) = 1$ (called the *restricted inverse law*)

  (b) $1/(1/x) = 1$ (called *reflection*)

A couple of interesting things follow from the meadow axioms. It can be shown that $1/0 = 0$. Perhaps more interestingly, it can be shown that $x \times 1/x = 1$ (when x is not 0) and $x \times 1/x = 0$ (when $x = 0$). In other words, in the non-zero fragment we recover the multiplicative inverse as with fields. This chapter extends Π with negative and fractional types in a manner that is consistent with meadows. The resulting language is called $\Pi^{\eta\epsilon}$.

However as *fields* are the dominant mathematical structure for rationals today, the rest of the presentation shall focus on the field-like fragment of the system. Thus we assume that when we speak of division, we are speaking of division by non-zero values. We will return to the caveat of non-zero division as appropriate at different points in the subsequent discussion. This section introduces the syntax and graphical languages for negatives and fractionals. The general outline is as follows:

(1) As established in Chapter 3, the underlying categorical semantics of our core reversible language Π is based on *two* distinct symmetric monoidal structures, one with $+$ as the monoidal tensor, and one with $\times$ as the monoidal tensor.

(2) We extend each underlying symmetric monoidal structure to a *compact closed* structure[1] by adding a dual for each object and two special morphisms[2] traditionally called η and ϵ.

After presenting the extension at the syntactic level, we discuss the categorical semantics informally via the graphical language. The operational semantics is presented in Sec. 3.

As will be detailed in the remainder of this section, the compact closed structure provides several properties of interest: (i) morphisms or wires are allowed to run from right to left, (ii) both monoids admit a trace that, depending on the situation, can be used to implement various notions of loops and recursion [Hasegawa, 1997, 2009, Joyal et al., 1996],

[1] We refer the reader to Selinger [2011] for the precise definitions.

[2] Note that when we say *compact closed* we are admittedly being sloppy here as discussed in Section 1.3.

(iii) the structure includes an isomorphism showing that the dual operator is an involution ($b \leftrightarrow 1/(1/b)$ and $b \leftrightarrow -(-b)$), and (iv) the structure is equipped with objects representing higher-order functions and a morphism *eval* that applies these functional objects. Interestingly each monoidal structure provides the same ingredients, resulting in two dualities, two traces, two involutions, and two notions of higher-order functions.

1.1. Syntax and Types. Describing the syntax and types of an extension to Π with additive and multiplicative duality is fairly straightforward. Basically, for the additive case, we extend the language with negative types denoted $-b$, negative values denoted $-v$, and two isomorphisms η^+ and ϵ^+. Similarly, for the multiplicative case, we extend the language with fractional types denoted $1/b$, fractional values denoted $1/v$, and two isomorphisms $\eta^\times$ and $\epsilon^\times$.

Definition 1.1. *(Syntax of $\Pi^{\eta\epsilon}$) The syntax of $\Pi^{\eta\epsilon}$ is obtained by extending the syntax of Π by the following syntactic forms.*

$$\begin{array}{rcl} \mathit{Value\ Types}, b & = & ... \mid -b \mid 1/b \\ \mathit{Values}, v & = & ... \mid -v \mid 1/v \\ \\ \mathit{Isomorphisms}, iso & = & ... \mid \eta^+ \mid \epsilon^+ \mid \eta^\times \mid \epsilon^\times \end{array}$$

The important side condition in the above is that division by 0 is not allowed, i.e. for all types of the form $1/b$, b must not be isomorphic to 0.

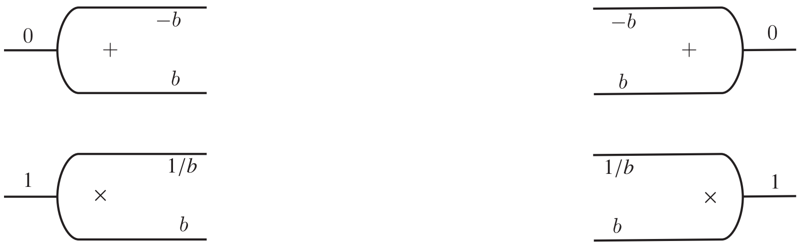
For convenience, we sometimes use the notations $b_1 - b_2$ and b_1/b_2 to indicate the types $b_1 + (-b_2)$ and $b_1 \times (1/b_2)$ respectively. The types of the new constructs are:

Definition 1.2. *(Type system for $\Pi^{\eta\epsilon}$) We extend the type judgments for primitive isomorphisms and values of Π to include the following:*

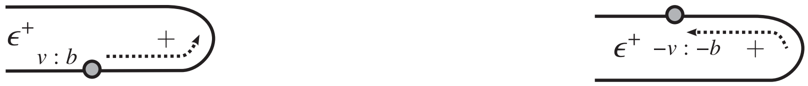
$$\begin{array}{llr} \eta^+ & : 0 \leftrightarrow (-b) + b : & \epsilon^+ \\ \eta^\times & : 1 \leftrightarrow (1/b) \times b : & \epsilon^\times \end{array}$$

$$\frac{\vdash v : b}{\vdash -v : -b} \quad \frac{\vdash v : b}{\vdash 1/v : 1/b}$$

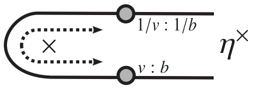
1.2. Graphical Language. For the graphical language, we visually represent η^{+}, ϵ^{+}, $\eta^{\times}$, and $\epsilon^{\times}$ as U-shaped connectors. On the left below are η^{+} and $\eta^{\times}$ showing the maps from 0 to $-b + b$ and 1 to $1/b \times b$. On the right are ϵ^{+} and $\epsilon^{\times}$ showing the maps from $-b + b$ to 0 and $1/b \times b$ to 1. Even though the diagrams below show the 0 and 1 wires for completeness, later diagrams will always drop them in contexts where they can be implicitly introduced and eliminated.

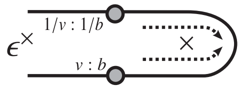


The usual interpretation of the type $b_1 + b_2$ is that we either have an appropriately tagged value of type b_1 or an appropriately tagged value of type b_2. This interpretation is maintained in the presence of η^{+} and ϵ^{+} in the following sense: a value of type $\mathit{right}\ v : -b + b$ flowing into an ϵ^{+} on the lower wire switches to a value $\mathit{left}\ (-v) : -b + b$ that flows backwards on the upper wire. The inversion of direction is captured by the negative sign on the value.



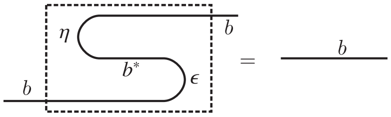
Similarly, in the usual interpretation of $b_1 \times b_2$ we have both a value of type b_1 and a value of type b_2. This interpretation is maintained in the presence of fractionals. Hence an $\eta^{\times} : 1 \leftrightarrow (1/b) \times b$ is to be viewed as a fission point for a value of type b and its multiplicative inverse $1/b$. As we will see in Sec. 3, this corresponds to the creation of two values α and $1/\alpha$ where α is a fresh logic variable. The operator $\epsilon^{\times}$ then becomes a unification site for these logic variables:





If the values being unified are not logic variables and they do not match, then the computation is undefined. This collapse is leveraged in the definition of the SAT solver of Section 5. Maintaining the analog with physics, this corresponds to the collapse of a waveform.

For a monoidal category to be compact closed the maps η and ϵ must satisfy a coherence condition that is usually visualized as follows:



where b^* represents the dual of b. In the case of negatives, the condition amounts to checking that reversing direction twice is a no-op. In the case of fractionals, the condition amounts to checking that creating values α and $1/\alpha$ and immediately unifying them is also a no-op.

1.3. Generalized Inverse Law and Compact Closed Categories. It must be noted that in the multiplicative world $\eta^\times : 1 \leftrightarrow (1/b) \times b : \epsilon\times$ hold only when b is not 0. In other words, this is not an isomorphism that is available at every type b. One must be careful to not violate the side condition in every construction that this combinator is used in. This is similar to doing proofs in algebra where one must take care to not divide by zero. For the constructions in the rest of the chapter it is indeed the case that we use $\eta^\times$ and $\epsilon^\times$ for non-zero values of b.

How does this connect to our previous requirement of using meadows? It does in the following sense: What we are using here is the definition of a field but with the extension that $1/0 = 0$. These are called ‘zero-totalized fields’ (ZTF). To be completely accurate we should also have included the morphism corresponding to $1/0 = 0$ in the presentation above. This may be added for the sake of completeness but we do not use this in the rest of the chapter. Meadows and zero-totalized fields are the same in the sense that they have the same equational theory (see Section 3.3 of Bergstra et al. [2009]). In other words, every equation that holds in one holds in the other.

Thus directly adding morphisms corresponding to the meadow axioms or carefully using the field axioms (ensuring that we don't violate their side conditions) gets us the same equations – and in the setting of $\Pi^{\eta\epsilon}$, gets us the same programs. In other words we could have added morphisms corresponding to *reflection* and the *restricted inverse law* directly:

(1) $\mathit{refll} : 1/(1/x) \leftrightarrow x : \mathit{reflr}$

(2) $\mathit{rill} : x \times (x \times (1/x)) \leftrightarrow x : \mathit{rilr}$

We have chosen to go with the ZTF style presentation of $\Pi^{\eta\epsilon}$ in this Chapter because having the $\eta^{\times}$ and $\epsilon^{\times}$ axioms directly is very convenient for (1) constructing higher-order functions, (2) constructing traces and (3) reusing other constructions and connections from the theory of compact closed categories. This convenience comes at the cost of ensuring that the side conditions are not violated in every single construction.

For a category to be compact closed there must be a dual object for every object b such that $\eta^{\times} : 1 \leftrightarrow 1/b{\times}b : \epsilon^{\times}$ morphisms hold. Hence strictly speaking the multiplicative monoid is not compact closed. However, as discussed above, these morphisms are applicable to all non-zero objects and one may say that $\Pi^{\eta\epsilon}$ is compact closed over the non-zero fragment for the multiplicative monoid. That said, this is a useful observation since it allows us leverage things like the trace construction in the sub-category of non-zero objects.

Finally, it must be noted that none of this applies to the additive monoid, $(+, 0)$. The additive monoid satisfies η^{+} and $\eta^{\times}$ for all objects.

1.4. Dualizing objects and collapse. The folklore around categorification of fields considers it a difficult task because much of the structure of interesting categories tend to collapse in the presence of dualizing objects. As with Π, $\Pi^{\eta\epsilon}$ does not have initial or terminal objects and neither does it have categorical products or coproducts (which correspond to information effects). If the category was equipped with a cartesian structure or with initial and terminal objects, then the structure would collapse. In other words, it can be shown that one could define a morphism from 0 to 1 and consequently from any object to any other object.

This notion of categorical collapse in the presence of duality is not well documented in the literature. The one instance of of this idea that we are aware of is Abramsky [2009]:

> [...] a well-known result by Joyal in categorical logic showing that a 'Boolean cartesian closed category' trivializes, which provides a major road-block to the computational interpretation of classical logic.

Later in the same paper this idea is expanded into the so called Joyal's lemma (see Sec 2.1 of Abramsky [2009]) which states:

> Any cartesian closed category with a dualizing object is a preorder (hence trivial as a semantics for proofs or computational processes).

1.5. On Counting. One important point that needs clarification is the following: Consider the map $\eta^{\times} : 1 \leftrightarrow (1/\mathit{bool}) \times \mathit{bool}$. How is there an information preserving map from 1, which has one value, to say $(1/\mathit{bool}) \times \mathit{bool}$ which has four distinct values?

In the simple world of Π it was indeed the case that every morphism mapped types inhabited by the same number of values to each other. To understand the situation here however, one must go back to Chapter 6 Section 1.2 where we see that information preserving maps can exist between types that are inhabited by differing number of values. In other words, information preserving functions are not a property of the types of the functions, but of the operational nature of the function.

In that Section however the maps were either bijective (when defined) or non-terminating. Non-termination was the unobservable. In this setting however, $\eta^{\times}$ is not a divergent computation – but has a similar restriction on what is observable. As we will see in the operational semantics, when $\eta^{\times}$ is given $()$, what we get as the result is not one of the four values inhabiting $1/\mathit{bool} \times \mathit{bool}$ but rather an entangled pair of type $1/\mathit{bool}$ and bool. Neither value is observable independently and whenever $\eta^{\times}$ is applied, the pair obtained is indistinguishable from any other such pair. An entangled pair can be thought of as a commitment to a choice, without revealing the actual choice. The observer has to do more work to see the value.

To observe the actual value of the *bool* component of this pair, the observer has to supply additional information. Say the observer supplies $1/\mathit{false}$ to the $1/\mathit{bool}$ (i.e. fixes one of the two possibilities), the entangled *bool* component is now grounded and reveals itself to be false. Thus, even though the observer of $\eta^{\times}$ is given an inhabitant of $1/\mathit{bool} \times \mathit{bool}$, further information has to supplied to observe which specific value has been obtained.

An analogous situation exists with the additive monoid wherein one component is negative and the other is positive. One has to supply a value to the negative component to observe the positive component. In terms of size of the types, values of the form $1/b$ are thought of as having fractional sizes since their role is to constrain larger values of the form $b{\times}b'$ to ones where the first component matches the constraint imposed by $1/b$. For example, if b and b' are $\mathit{bool}{\times}\mathit{bool}$ and bool respectively, and the supplied constraint is $1/(\mathit{false}, \mathit{false})$, then not all values $(\mathit{bool} \times \mathit{bool}) \times \mathit{bool}$ are allowed – only those with the first component $(\mathit{false}, \mathit{false})$ are allowed. This effectively restricts the choices to $((\mathit{false}, \mathit{false}), \mathit{true})$ and $((\mathit{false}, \mathit{false}), \mathit{false})$, thus limiting the size of $(\mathit{bool}{\times}\mathit{bool}){\times}\mathit{bool}$ to 2, given a $1/(\mathit{bool}{\times}\mathit{bool})$ constraint. Again, an analogous situation applies to negative types whose types are interpreted to be negative.

2. Small-Step Semantics for Π

To give $\Pi^{\eta\epsilon}$ an operational semantics, we need a setting where we can talk about the backward flow of particles on the wires. This is best done using a small-step semantics. Hence we shall first proceed to develop a small-step semantics for Π and then extend it appropriately to express negatives and fractionals.

The semantics of the primitive combinators is given by the following single-step reductions. Since there are no values of type 0, the rules omit the impossible cases:

Definition 2.1. *(Small-step reductions for Primitive Isomorphisms)*

$$\mathit{swap}^{+}\ (\mathit{left}\ v) \quad \mapsto \quad \mathit{right}\ v$$

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

The reductions for the primitive isomorphisms above are exactly the same as have been presented before in Chapter 3. The reductions for the closure combinators are however presented in a small-step operational style using the following definitions of evaluation contexts and machine states:

Definition 2.2. *(Syntax of Evaluation Contexts and Machine States)*

$$\begin{array}{rcl}
\textit{Combinator Contexts}, C & = & \Box \mid \textit{Fst}\ C\ c \mid \textit{Snd}\ c\ C \\
& \mid & L^{\times}\ C\ c\ v \mid R^{\times}\ c\ v\ C \\
& \mid & L^{+}\ C\ c \mid R^{+}\ c\ C \\
\textit{Machine states} & = & \langle c, v, C\rangle \mid [c, v, C] \\
\textit{Start state} & = & \langle c, v, \Box\rangle \\
\textit{Stop State} & = & [c, v, \Box]
\end{array}$$

The machine transitions below track the flow of particles through a circuit. The start machine state, $\langle c, v, \Box\rangle$, denotes the particle v about to be evaluated by the circuit c. The end machine state, $[c, v, \Box]$, denotes the situation where the particle v has exited the circuit c.

Definition 2.3. *(Small-step operational semantics for* Π *)*

$$\begin{array}{rcll}
\langle \textit{iso}, v, C\rangle & \mapsto & [\textit{iso}, v', C] \quad \textit{where iso } v \mapsto v' & (1) \\
\langle c_1 \fatsemi c_2, v, C\rangle & \mapsto & \langle c_1, v, \textit{Fst}\ C\ c_2\rangle & (2) \\
[c_1, v, \textit{Fst}\ C\ c_2] & \mapsto & \langle c_2, v, \textit{Snd}\ c_1\ C\rangle & (3) \\
[c_2, v, \textit{Snd}\ c_1\ C] & \mapsto & [c_1 \fatsemi c_2, v, C] & (4) \\
\langle c_1 + c_2, \textit{left}\ v, C\rangle & \mapsto & \left\langle c_1, v, L^{+}\ C\ c_2\right\rangle & (5) \\
[c_1, v, L^{+}\ C\ c_2] & \mapsto & [c_1 + c_2, \textit{left}\ v, C] & (6) \\
\langle c_1 + c_2, \textit{right}\ v, C\rangle & \mapsto & \left\langle c_2, v, R^{+}\ c_1\ C\right\rangle & (7) \\
[c_2, v, R^{+}\ c_1\ C] & \mapsto & [c_1 + c_2, \textit{right}\ v, C] & (8) \\
\langle c_1 \times c_2, (v_1, v_2), C\rangle & \mapsto & \left\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2\right\rangle & (9) \\
[c_1, v_1, L^{\times}\ C\ c_2\ v_2] & \mapsto & \left\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C\right\rangle & (10) \\
[c_2, v_2, R^{\times}\ c_1\ v_1\ C] & \mapsto & [c_1 \times c_2, (v_1, v_2), C] & (11)
\end{array}$$

The above rules accomplish the same at the big-step operational semantics of Chapter 3. Rule (1) describes evaluation by a primitive isomorphism. Rules (2), (3) and (4) deal with sequential evaluation. Rule (2) says that for the value v to flow through the sequence $c_1 \fatsemi c_2$, it should first flow through c_1 with c_2 pending in the context $(Fst\ C\ c_2)$. Rule (3) says the value v that exits from c_1 should proceed to flow through c_2. Rule (4) says that when the value v exits c_2, it also exits the sequential composition $c_1 \fatsemi c_2$. Rules (5) to (8) deal with $c_1 + c_2$ in the same way. In the case of sums, the shape of the value, i.e., whether it is tagged with $left$ or $right$, determines whether path c_1 or path c_2 is taken. Rules (9), (10) and (11) deal with $c_1 \times c_2$ similarly. In the case of products the value should have the form (v_1, v_2) where v_1 flows through c_1 and v_2 flows through c_2. Both these paths are entirely independent of each other and we could evaluate either first, or evaluate both in parallel. In this presentation we have chosen to follow c_1 first, but this choice is entirely arbitrary.

The interesting thing about the semantics is that it represents a reversible abstract machine. In other words, we can compute the start state from the stop state by changing the reductions $\mapsto$ to run backwards $\hookleftarrow$. When running backwards, we use the isomorphism represented by a combinator c in the reverse direction, i.e., we use the adjoint $c^\dagger$.

Proposition 2.4 (Logical Reversibility). $\langle c, v, \Box \rangle \mapsto [c, v', \Box]$ *iff* $\left\langle c^\dagger, v', \Box \right\rangle \mapsto [c^\dagger, v, \Box]$

3. Small-Step Semantics for $\Pi^{\eta\epsilon}$

In this section we develop an operational semantics for Π extended with negative and fractional types, $\Pi^{\eta\epsilon}$. The operational semantics takes the abstract categorical diagrams which were previously known but gives them a computational interpretation based on reverse execution for negative types and unification for fractional types.

Even though this computational interpretation appears straightforward in retrospect, it constitutes a breakthrough because it breaks the preconceived idea that there is only one notion of duality and hence that both negative and fractional types should be implemented using the same underlying mechanism. In some sense, the folklore literature on monoidal categories describing the evaluations for additive and multiplicative fragments are “particle-style” and “wave-style” can be seen to suggest either a unified or separate implementations. But, since de Morgan, we have been predisposed to think of only one notion of duality which persisted to even linear logic. (See Chapter 9 Section 3 for further discussion of duality.)

For the purposes of this section, we start with the semantics for Π defined in Section 2, and we will systematically rewrite it to achieve the desired $\Pi^{\eta\epsilon}$ semantics. This rewriting consists of two main steps:

(1) We rewrite the semantics in Sec 2 using unification of the values instead of direct pattern matching. This gives us the necessary infrastructure for fractional types. We assume the reader is familiar with the idea of unification as realized using logic variables, substitutions, and reification as presented in any standard text on logic programming.
(2) We write a reverse interpreter by reversing the direction of the $\mapsto$ reductions. This gives us the necessary infrastructure for negative types.

3.1. Unification. Previously the reductions of the primitive isomorphisms were specified in the following form: $iso\ v_{input\ pattern} \mapsto v_{output\ pattern}$
Instead of relying on pattern-matching, we rewrite the rules to accept an incoming substitution to which the required pattern-matching rules are added as constraints. The general case is: $iso\ s\ v' \mapsto (v_{output\ pattern}, s[v' \approx v_{input\ pattern}])$

Using the same idea, the entire semantics can be extended to thread the substitution as shown below:

$$
\begin{array}{rcl}
\langle iso, v, C, s\rangle_{\triangleright} & \mapsto & [iso, v', C, s']_{\triangleright} \\
 & & where\ iso\ s\ v \mapsto (v', s') \\
\langle c_1 \fatsemi c_2, v, C, s\rangle_{\triangleright} & \mapsto & \langle c_1, v, Fst\ C\ c_2, s\rangle_{\triangleright} \\
[c_1, v, Fst\ C\ c_2, s]_{\triangleright} & \mapsto & \langle c_2, v, Snd\ c_1\ C, s\rangle_{\triangleright} \\
[c_2, v, Snd\ c_1\ C, s]_{\triangleright} & \mapsto & [c_1 \fatsemi c_2, v, C, s]_{\triangleright} \\
\left\langle c_1 + c_2, v', C, s\right\rangle_{\triangleright} & \mapsto & \left\langle c_1, v, L^+\ C\ c_2, s[v' \approx left\ v]\right\rangle_{\triangleright} \\
[c_1, v, L^+\ C\ c_2, s]_{\triangleright} & \mapsto & [c_1 + c_2, left\ v, C, s]_{\triangleright}
\end{array}
$$

$$\begin{aligned}
\left\langle c_1 + c_2, v', C, s \right\rangle_{\triangleright} &\mapsto \left\langle c_2, v, R^{+}\ c_1\ C, s[v' \approx \textit{right}\ v] \right\rangle_{\triangleright} \\
[c_2, v, R^{+}\ c_1\ C, s]_{\triangleright} &\mapsto [c_1 + c_2, \textit{right}\ v, C, s]_{\triangleright} \\
\left\langle c_1 \times c_2, v', C, s \right\rangle_{\triangleright} &\mapsto \left\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2, s' \right\rangle_{\triangleright} \\
& \quad \textit{where}\ s' = s[v \approx (v_1, v_2)] \\
[c_1, v_1, L^{\times}\ C\ c_2\ v_2, s]_{\triangleright} &\mapsto \left\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C, s \right\rangle_{\triangleright} \\
[c_2, v_2, R^{\times}\ c_1\ v_1\ C, s]_{\triangleright} &\mapsto [c_1 \times c_2, (v_1, v_2), C, s]_{\triangleright}
\end{aligned}$$

Most of the rules look obviously correct but there is a subtle point. Consider the case for $c_1 + c_2$. Previously the shape of the value determined which of the combinators c_1 or c_2 to use. Now the incoming value could be a fresh logical variable, and indeed we have two rules with the same left hand side, and only depending on the success or failure of some further unification, can we decide which of them applies. The situation is common in logic programming in the sense that it is considered a non-deterministic process that keeps searching for the right substitution if any. Typical implementations of logic programming languages further provide top-level mechanisms to manipulate this non-deterministic search, for example by returning one answer and giving the user the option to ask for more answers. We abstract from this top-level semantics and view the rules above as being applied non-deterministically.

3.2. Reverse Execution. Given that our language is reversible (Prop. 2.4), a backward evaluator is relatively straightforward to implement: using the backward evaluator to calculate $c\ v$ is equivalent $c^{\dagger}\,v$ in the forward evaluator.

$$\begin{aligned}
[\textit{iso}, v, C, s]_{\triangleleft} &\mapsto \left\langle \textit{iso}, v', C, s' \right\rangle_{\triangleleft} \\
& \quad \textit{where}\ \textit{iso}^{\dagger}\, s\ v \mapsto (v', s') \\
\langle c_1, v, \textit{Fst}\ C\ c_2, s \rangle_{\triangleleft} &\mapsto \langle c_1 \fatsemi c_2, v, C, s \rangle_{\triangleleft} \\
\langle c_2, v, \textit{Snd}\ c_1\ C, s \rangle_{\triangleleft} &\mapsto [c_1, v, \textit{Fst}\ C\ c_2, s]_{\triangleleft}
\end{aligned}$$

$$\begin{aligned}
[c_1 \fatsemi c_2, v, C, s]_{\triangleleft} &\mapsto [c_2, v, \mathit{Snd}\ c_1\ C, s]_{\triangleleft} \\
\left\langle c_1, v, L^{+}\ C\ c_2, s \right\rangle_{\triangleleft} &\mapsto \langle c_1 + c_2, \mathit{left}\ v, C \rangle_{\triangleleft} \\
[c_1 + c_2, v', C, s]_{\triangleleft} &\mapsto [c_1, v, L^{+}\ C\ c_2, s[v' \approx \mathit{left}\ v]]_{\triangleleft} \\
\left\langle c_2, v, R^{+}\ c_1\ C, s \right\rangle_{\triangleleft} &\mapsto \langle c_1 + c_2, \mathit{right}\ v, C \rangle_{\triangleleft} \\
[c_1 + c_2, v', C, s]_{\triangleleft} &\mapsto [c_2, v, R^{+}\ c_1\ C, s[v' \approx \mathit{right}\ v]]_{\triangleleft} \\
\left\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2, s \right\rangle_{\triangleleft} &\mapsto \langle c_1 \times c_2, (v_1, v_2), C, s \rangle_{\triangleleft} \\
\left\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C, s \right\rangle_{\triangleleft} &\mapsto [c_1, v_1, L^{\times}\ C\ c_2\ v_2, s]_{\triangleleft} \\
[c_1 \times c_2, v, C, s]_{\triangleleft} &\mapsto [c_2, v_2, R^{\times}\ c_1\ v_1\ C, S']_{\triangleleft} \\
&\quad\ \mathit{where}\ s' = s[v \approx (v_1, v_2)]
\end{aligned}$$

3.3. Semantics of $\Pi^{\eta\epsilon}$. Combining the two constructions above, we get the semantics of the full $\Pi^{\eta\epsilon}$ language by adding the rules for the two variants of η and ϵ.

(1) To add multiplicative duality, we add the following rules to the set of primitive isomorphisms:

   (a) $\eta^{\times} s\ () \mapsto ((1/v, v), s)$ where v is a fresh logic variable.

   As explained previously $\eta^{\times}$ creates two values $1/v$ and $1/v$ from a single logic variable.

   (b) $\epsilon^{\times} s\ v \mapsto (1, s[v \approx (1/v', v')])$ where v' is a fresh logic variable.

   Similarly, $\epsilon^{\times}$ unifies the values on incoming wires. The incoming value v represents the values of both wires and hence unifying them is accomplished in terms of an intermediate logic variable v'.

(2) To add negative types we add the following rules to the reductions above. The additions formalize our previous discussions and should not be surprising at this point.

   (a) The rules for ϵ^{+} essentially transfer control from the forward evaluator (whose states are tagged by $\triangleright$) to the backward evaluator (whose states are tagged by $\triangleleft$). In other words, after an ϵ^{+} the direction of the world is reversed. The pattern matching done by the unification ensures that a value on the $right$ wire is tagged to be negative and transferred to the $left$ wire, and vice versa.

$$\langle \epsilon^+, v, C, s \rangle_{\triangleright} \mapsto \langle \epsilon^+, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v'] \rangle_{\triangleleft}$$
$$\langle \epsilon^+, v, C, s \rangle_{\triangleright} \mapsto \langle \epsilon^+, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')] \rangle_{\triangleleft}$$

Note that there is no evaluation rule for η^+ in the forward evaluator. This corresponds to the fact that there is no value of type 0 and hence the forward evaluator can never execute an η^+.

(b) The rules for η^+ are added to the backward evaluator. A program executing backwards starts executing forwards after the execution of the η^+. Dual to the previous case, there is no rule for ϵ^+ in the backward evaluator since the output type of ϵ^+ is 0.

$$[\eta^+, v, C, s]_{\triangleleft} \mapsto [\eta^+, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v']]_{\triangleright}$$
$$[\eta^+, v, C, s]_{\triangleleft} \mapsto [\eta^+, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')]]_{\triangleright}$$

Definition 3.1. *(Small-step operational Semantics of $\Pi^{\eta\epsilon}$)*

Primitive Isomorphisms (including $\eta^\times$ and $\epsilon^\times$)

$$\begin{aligned}
\mathit{swap}^+\ s\ \ v &\ \mapsto\ (\mathit{right}\ v', s[v \approx (\mathit{left}\ v')]) \\
\mathit{swap}^+\ s\ \ v &\ \mapsto\ (\mathit{left}\ v', s[v \approx (\mathit{right}\ v')]) \\
\mathit{assocl}^+\ s\ \ v &\ \mapsto\ (\mathit{left}\ (\mathit{left}\ v_1), s[v \approx (\mathit{left}\ v_1)]) \\
\mathit{assocl}^+\ s\ \ v &\ \mapsto\ (\mathit{left}\ (\mathit{right}\ v_2), s[v \approx (\mathit{right}\ (\mathit{left}\ v_2))]) \\
\mathit{assocl}^+\ s\ \ v &\ \mapsto\ (\mathit{right}\ v_3, s[v \approx (\mathit{right}\ (\mathit{right}\ v_3))]) \\
\mathit{assocr}^+\ s\ \ v &\ \mapsto\ (\mathit{left}\ v_1, s[v \approx (\mathit{left}\ (\mathit{left}\ v_1))]) \\
\mathit{assocr}^+\ s\ \ v &\ \mapsto\ (\mathit{right}\ (\mathit{left}\ v_2), s[v \approx (\mathit{left}\ (\mathit{right}\ v_2))]) \\
\mathit{assocr}^+\ s\ \ v &\ \mapsto\ (\mathit{right}\ (\mathit{right}\ v_3), s[v \approx (\mathit{right}\ v_3)]) \\
\mathit{unite}\ s\ \ v &\ \mapsto\ (v', s[v \approx ((), v')]) \\
\mathit{uniti}\ s\ \ v &\ \mapsto\ (((), v), s)
\end{aligned}$$

$$\begin{aligned}
\mathit{swap}^{\times}\ s\ \ v &\ \mapsto\ ((v_2, v_1), s[v \approx (v_1, v_2)]) \\
\mathit{assocl}^{\times}\ s\ \ v &\ \mapsto\ (((v_1, v_2), v_3), s[v \approx (v_1, (v_2, v_3))]) \\
\mathit{assocr}^{\times}\ s\ \ v &\ \mapsto\ ((v_1, (v_2, v_3)), s[v \approx ((v_1, v_2), v_3)]) \\
\mathit{distrib}\ \ s\ \ v &\ \mapsto\ (\mathit{left}\ (v_1, v_3), s[v \approx (\mathit{left}\ v_1, v_3)]) \\
\mathit{distrib}\ \ s\ \ v &\ \mapsto\ (\mathit{right}\ (v_2, v_3), s[v \approx (\mathit{right}\ v_2, v_3)]) \\
\mathit{factor}\ \ s\ \ v &\ \mapsto\ ((\mathit{left}\ v_1, v_3), s[v \approx (\mathit{left}\ (v_1, v_3))]) \\
\mathit{factor}\ \ s\ \ v &\ \mapsto\ ((\mathit{right}\ v_2, v_3), s[v \approx (\mathit{right}\ (v_2, v_3))]) \\
\eta^{\times}\ s\ \ v &\ \mapsto\ ((1/v', v'), s[v \approx ()]) \\
\epsilon^{\times}\ s\ \ v &\ \mapsto\ (1, s[v \approx (1/v', v')])
\end{aligned}$$

In the above rules, v', v_1, v_2 and v_3 are fresh logic variables.

Forward execution:

$$\begin{aligned}
\langle \mathit{iso}, v, C, s\rangle_{\triangleright} &\ \mapsto\ [\mathit{iso}, v', C, s']_{\triangleright} \\
&\ \ \ \mathit{where}\ \mathit{iso}\ s\ v \mapsto (v', s') \\
\langle c_1 \fatsemi c_2, v, C, s\rangle_{\triangleright} &\ \mapsto\ \langle c_1, v, \mathit{Fst}\ C\ c_2, s\rangle_{\triangleright} \\
[c_1, v, \mathit{Fst}\ C\ c_2, s]_{\triangleright} &\ \mapsto\ \langle c_2, v, \mathit{Snd}\ c_1\ C, s\rangle_{\triangleright} \\
[c_2, v, \mathit{Snd}\ c_1\ C, s]_{\triangleright} &\ \mapsto\ [c_1 \fatsemi c_2, v, C, s]_{\triangleright} \\
\left\langle c_1 + c_2, v', C, s\right\rangle_{\triangleright} &\ \mapsto\ \left\langle c_1, v, L^{+}\ C\ c_2, s[v' \approx \mathit{left}\ v]\right\rangle_{\triangleright} \\
[c_1, v, L^{+}\ C\ c_2, s]_{\triangleright} &\ \mapsto\ [c_1 + c_2, \mathit{left}\ v, C, s]_{\triangleright} \\
\left\langle c_1 + c_2, v', C, s\right\rangle_{\triangleright} &\ \mapsto\ \left\langle c_2, v, R^{+}\ c_1\ C, s[v' \approx \mathit{right}\ v]\right\rangle_{\triangleright} \\
[c_2, v, R^{+}\ c_1\ C, s]_{\triangleright} &\ \mapsto\ [c_1 + c_2, \mathit{right}\ v, C, s]_{\triangleright} \\
\left\langle c_1 \times c_2, v', C, s\right\rangle_{\triangleright} &\ \mapsto\ \left\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2, s' = s[v \approx (v_1, v_2)]\right\rangle_{\triangleright} \\
[c_1, v_1, L^{\times}\ C\ c_2\ v_2, s]_{\triangleright} &\ \mapsto\ \left\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C, s\right\rangle_{\triangleright}
\end{aligned}$$

$$\begin{array}{rcl}
[c_2, v_2, R^{\times}\ c_1\ v_1\ C, s]_{\triangleright} & \mapsto & [c_1 \times c_2, (v_1, v_2), C, s]_{\triangleright} \\
\left\langle \epsilon^{+}, v, C, s \right\rangle_{\triangleright} & \mapsto & \left\langle \epsilon^{+}, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v'] \right\rangle_{\triangleleft} \\
\left\langle \epsilon^{+}, v, C, s \right\rangle_{\triangleright} & \mapsto & \left\langle \epsilon^{+}, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')] \right\rangle_{\triangleleft}
\end{array}$$

Reverse Execution:

$$\begin{array}{rcl}
[\mathit{iso}, v, C, s]_{\triangleleft} & \mapsto & \left\langle \mathit{iso}, v', C, s' \right\rangle_{\triangleleft} \\
 & & \mathit{where}\ \mathit{iso}^{\dagger}\ s\ v \quad \mapsto\ (v', s') \\
\langle c_1, v, \mathit{Fst}\ C\ c_2, s \rangle_{\triangleleft} & \mapsto & \langle c_1 \fatsemi c_2, v, C, s \rangle_{\triangleleft} \\
\langle c_2, v, \mathit{Snd}\ c_1\ C, s \rangle_{\triangleleft} & \mapsto & [c_1, v, \mathit{Fst}\ C\ c_2, s]_{\triangleleft} \\
[c_1 \fatsemi c_2, v, C, s]_{\triangleleft} & \mapsto & [c_2, v, \mathit{Snd}\ c_1\ C, s]_{\triangleleft} \\
\left\langle c_1, v, L^{+}\ C\ c_2, s \right\rangle_{\triangleleft} & \mapsto & \langle c_1 + c_2, \mathit{left}\ v, C \rangle_{\triangleleft} \\
[c_1 + c_2, v', C, s]_{\triangleleft} & \mapsto & [c_1, v, L^{+}\ C\ c_2, s[v' \approx \mathit{left}\ v]]_{\triangleleft} \\
\left\langle c_2, v, R^{+}\ c_1\ C, s \right\rangle_{\triangleleft} & \mapsto & \langle c_1 + c_2, \mathit{right}\ v, C \rangle_{\triangleleft} \\
[c_1 + c_2, v', C, s]_{\triangleleft} & \mapsto & [c_2, v, R^{+}\ c_1\ C, s[v' \approx \mathit{right}\ v]]_{\triangleleft} \\
\left\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2, s \right\rangle_{\triangleleft} & \mapsto & \langle c_1 \times c_2, (v_1, v_2), C, s \rangle_{\triangleleft} \\
\left\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C, s \right\rangle_{\triangleleft} & \mapsto & [c_1, v_1, L^{\times}\ C\ c_2\ v_2, s]_{\triangleleft} \\
[c_1 \times c_2, v, C, s]_{\triangleleft} & \mapsto & [c_2, v_2, R^{\times}\ c_1\ v_1\ C, s[v \approx (v_1, v_2)]]_{\triangleleft} \\
[\eta^{+}, v, C, s]_{\triangleleft} & \mapsto & [\eta^{+}, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v']]_{\triangleright} \\
[\eta^{+}, v, C, s]_{\triangleleft} & \mapsto & [\eta^{+}, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')]]_{\triangleright}
\end{array}$$

Observability. The reductions above allow us to apply a program $c : b_1 \leftrightarrow b_2$ to an input $v_1 : b_1$ to produce a result $v_2 : b_2$ on termination. Execution is well defined only if b_1 and b_2 are entirely positive types. If either b_1 or b_2 is a negative or fractional type, the system has "dangling" unsatisfied demands or constraints. For this reason, we constrain entire programs to have positive non-fractional types. This is similar to the constraint that Zeilberger imposes to explain intuitionistic polarity and delimited control [Zeilberger, 2010].

3.4. Meadow Axioms. Alternately one could have added operational semantics for the meadow axioms. This would involve replacing the rules for $\eta^{\times}$ and $\epsilon^{\times}$ with the following rules:

(1) Restricted Inverse law:
   (a) $\mathit{rill}\ s\ v \mapsto (v_1, s[v \approx (v_1, (v_2, 1/v_2))])$ where v_1 and v_2 are fresh logic variables.
   (b) $\mathit{rilr}\ s\ v_1 \mapsto (v, s[v \approx (v_1, (v_2, 1/v_2))])$ where v and v_2 are fresh logic variables.

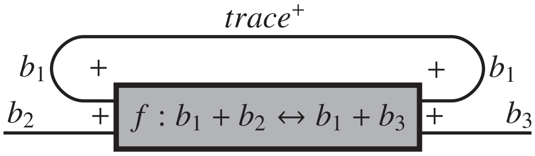
(2) Reflection:
   (a) $\mathit{refll}\ s\ v' \mapsto (v, s[v' \approx 1/(1/v)])$ where v is a fresh logic variable.
   (b) $\mathit{reflr}\ s\ v \mapsto (v', s[v' \approx 1/(1/v)])$ where v' is a fresh logic variable.

4. Expressiveness of $\Pi^{\eta\epsilon}$

We now review several interesting constructions related to looping, involution, and higher order functions. All the constructions below are standard: they are collected from Selinger's survey paper on monoidal categories [Selinger, 2011] and presented in the context of our language.

4.1. Trace. Every compact closed category admits a trace. For the additive case, we get the following definition. Given $f : b_1 + b_2 \leftrightarrow b_1 + b_3$, define $\mathit{trace}^{+} f : b_2 \leftrightarrow b_3$ as:

$$\mathit{trace}^{+} f = \mathit{zeroi} \fatsemi (\mathit{id} + \eta^{+}) \fatsemi (f + \mathit{id}) \fatsemi (\mathit{id} + \epsilon^{+}) \fatsemi \mathit{zeroe}$$



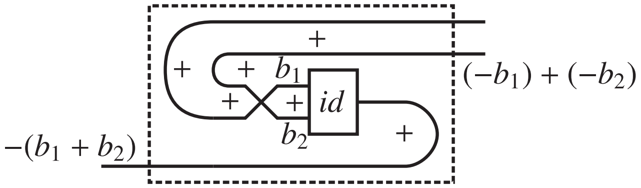
We have omitted some of the commutativity and associativity shuffling to communicate the main idea. We are given a value of type b_2 which we embed into $0 + b_2$ and then $(-b_1 + b_1) + b_2)$. This can be re-associated into $-b_1 + (b_1 + b_2)$. The component $b_1 + b_2$, which until now is just an appropriately tagged value of type b_2, is transformed to a value of type $b_1 + b_3$ by f. If the result is in the b_3-summand, it is produced as the answer; otherwise the result is in the b_1-summand; ϵ^+ is used to make it flow backwards to be fed to the η^+ located at the beginning of the sequence. Iteration continues until a b_3 is produced.

4.2. Involution (Principium Contradictiones). In a symmetric compact closed category, we can build isomorphisms that the dual operation is an involution. Specifically, we get the isomorphisms $b \leftrightarrow -(-b)$ and $b \leftrightarrow (1/(1/b))$. For the additive case, the isomorphism is defined as follows:

$(id + \eta^+) \fatsemi (swap^+ + id) \fatsemi (id + \epsilon^+)$

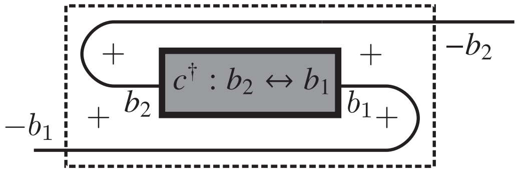
where we have omitted the 0 introduction and elimination. The idea is as follows: we start with a value of type b, embed it into $b + 0$ and use η to create something of type $b + (-(-b) + (-b))$. This is possible because η has the polymorphic type $-a + a$ which can be instantiated to $-b$. We then reshuffle the type to produce $-(-b) + (-b + b)$ and cancel the right hand side using ϵ^+. The construction for the multiplicative case is identical and omitted.

4.3. Duality preserves the monoidal tensor. As with compact closed categories, the dual on the objects distributes over the tensor. In terms of $\Pi^{\eta\epsilon}$ we have that $-(b_1 + b_2)$ can be mapped to $(-b_1) + (-b_2)$ and that $1/(b_1 \times b_2)$ can be mapped to $(1/b_1) \times (1/b_2)$. The isomorphism $-(b_1 + b_2) \leftrightarrow (-b_1) + (-b_2)$ can be realized as follows:



The multiplicative construction is similar.

4.4. Duality is a functor. Duality in $\Pi^{\eta\epsilon}$ can map objects to their duals and morphisms to act on dual objects. In other $c : b_1 \leftrightarrow b_2$ to $neg\ c : -b_1 \leftrightarrow -b_2$ in the additive monoid and to $inv\ c : 1/b_1 \leftrightarrow 1/b_2$ in the multiplicative monoid. The idea is simply to reverse the flow of values and use the adjoint of the operation:

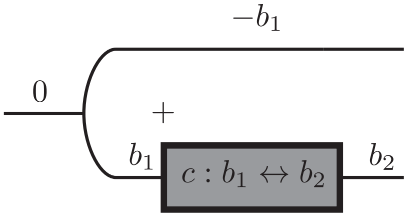


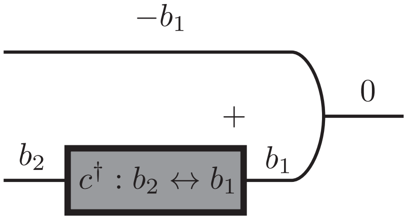
$$\frac{c : b_1 \leftrightarrow b_2}{neg\ c : (-b_1) \leftrightarrow (-b_2)}$$

This construction relies on the fact that every $\Pi^{\eta\epsilon}$ morphism has an adjoint. The inv construction is similar.

4.5. Functions and Delimited Continuations. Although these constructions are also standard, they are less known and they are particularly important in our context: we devote a little more time to explain them. Our discussion is mostly based on Abramsky and Coecke's article on categorical quantum mechanics [Abramsky and Coecke, 2008].

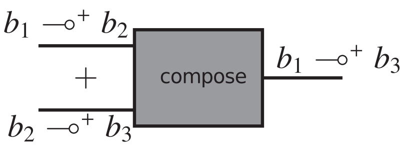
In a compact closed category, each morphism $f : b_1 \leftrightarrow b_2$ can be given a *name* and a *coname*. For the additive fragment, the name $\ulcorner f \urcorner$ has type $0 \leftrightarrow (-b_1 + b_2)$ and the coname $\llcorner f \lrcorner$ has type $b_1 + (-b_2) \leftrightarrow 0$. For the multiplicative fragment, the name $\ulcorner f \urcorner$ has type $1 \leftrightarrow ((1/b_1) \times b_2))$ and the coname $\llcorner f \lrcorner$ has type $(b_1 \times (1/b_2)) \leftrightarrow 1$. Intuitively, this means that for each morphism, it is possible to construct, from "nothing," an object in the category that denotes this morphism, and dually it is possible to eliminate this object. The construction of the name and coname of $c : b_1 \leftrightarrow b_2$ in the additive case can be visualized as follows:

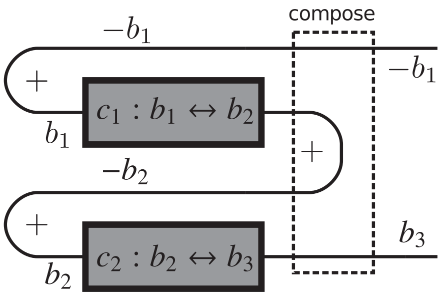




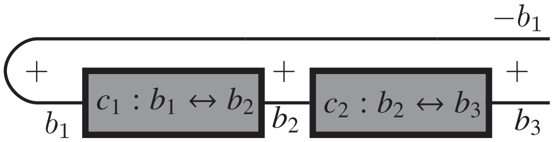
Intuitively the name consists of viewing c as a function and the coname consists of viewing c as a delimited continuation.

In addition to being able to represent morphisms, it is possible to express function composition. For the additive case, the composition is depicted below:

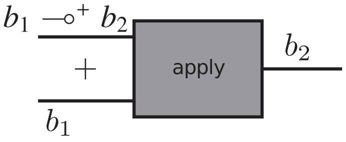


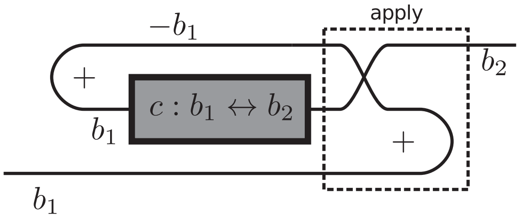

compose
$-b_1$
$-b_1$
$+$
$c_1 : b_1 \leftrightarrow b_2$
b_1
$+$
$-b_2$
$+$
$c_2 : b_2 \leftrightarrow b_3$
b_2
b_3

which is essentially equivalent to sequencing both the computation blocks as shown below:


$-b_1$
$+$
$c_1 : b_1 \leftrightarrow b_2$
$+$
$c_2 : b_2 \leftrightarrow b_3$
$+$
b_1
b_2
b_3

Applying a function to an argument consists of making the argument flow backwards to satisfy the demand of the function:





Having reviewed the representation of functions, we now discuss the similarities and differences between the two notions of functions and their relation to conventional (linear) functions which mix additive and multiplicative components. For that purpose, we use a small example. Consider a datatype *color* $= R|G|B$, and let us consider the following manipulations:

- Using the fact that 1 is the multiplicative unit, generate from the input $()$ the value $((),())$ of type 1×1;

- Apply the isomorphism $1 \leftrightarrow (1/b) \times b$ in parallel to each of the components of the above tuple. The resulting value is $((1/\alpha_1, \alpha_1), (1/\alpha_2, \alpha_2))$ where α_1 and α_2 are fresh logic variables;
- Using the fact that $\times$ is associative and commutative, we can rearrange the above tuple to produce the value:
  $((1/\alpha_1, \alpha_2), (1/\alpha_2, \alpha_1))$.

At this point we have constructed a strange mix of two $b \multimap^{\times} b$ functions; inputs of one function manifest themselves as outputs of the other. If $(1/\alpha_1, \alpha_2)$ is held by one subcomputation and $(1/\alpha_2, \alpha_1)$ is held by another subcomputation, these remixed functions form a communication channel between the two concurrent subcomputations. Unifying $1/\alpha_1$ with *color* R in one subcomputation, fixes α_1 to be R in the other. The type b thus takes the role of the type of the communication channel, indicating how much information can be communicated between the two subcomputations. Depending on the choice of the type b, an arbitrary number of bits may be communicated.

Dually, the additive reading of the above manipulations correspond to functions of the form $b \multimap^{+} b$, witnessing isomorphisms of the form $0 \leftrightarrow (-b) + b)$. The remixed additive functions express control flow transfer between two subcomputations, *only one of which exists* at any point, i.e., they capture the essence of coroutines.

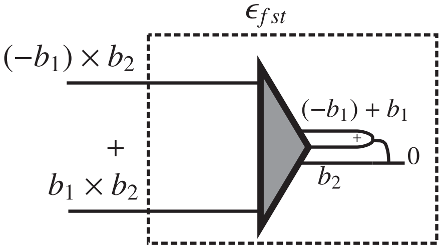
It should be evident that in a universe in which information is not guaranteed to be preserved by the computational infrastructure, the above slicing and dicing of functions would make no sense. But linearity is not sufficient: one must also recognize that the additive and multiplicative spaces are different.

4.6. Additional Constructions. The additional constructions below (presented with minimal commentary) confirm that conventional algebraic manipulations in the mathematical field of rationals do indeed correspond to realizable type isomorphisms in our setting. The constructions involving both negative and fractional types are novel.

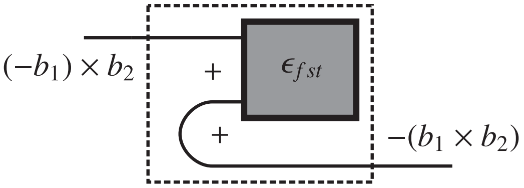
4.6.1. *Lifting negation out of* $\times$. The isomorphisms below state that the direction is *relative*. If b_1 and b_2 are flowing opposite to each other then it doesn't matter which direction is forwards and which is backwards. More interestingly as b_1 is moving backwards, it can "see the past" of b_2 which is equivalent to both particles moving backwards.

$(-b_1) \times b_2 \leftrightarrow -(b_1 \times b_2) \leftrightarrow b_1 \times (-b_2)$

To build these isomorphisms, we first build an intermediate construction which we call $\epsilon_{fst} : (-b_1) \times b_2 + b_1 \times b_2 \leftrightarrow 0$.



The isomorphism $(-b_1) \times b_2 \leftrightarrow -(b_1 \times b_2)$ can be constructed in terms of ϵ_{fst} as shown below.



The second isomorphism can be built in the same way by merely swapping the arguments.

4.6.2. *Multiplying Negatives.* $b_1 \times b_2 \leftrightarrow (-b_1) \times (-b_2)$

This isomorphism is a consequence of the fact that $-$ is an involution: it corresponds to the algebraic manipulation:

$b_1 \times b_2 = -(-(b_1 \times b_2)) = -((-b_1) \times b_2) = (-b_1) \times (-b_2)$

4.6.3. *Multiplying and Adding Fractions.* An isomorphism witnessing:

$b_1/b_2 \times b_3/b_4 \leftrightarrow (b_1 \times b_3)/(b_2 \times b_4)$

is straightforward. More surprisingly, it is also possible to construct isomorphisms witnessing:

$b_1/b + b_2/b \leftrightarrow (b_1 + b_2)/b$

$b_1/b_2 + b_3/b_4 \leftrightarrow (b_1 \times b_4 + b_3 \times b_2)/(b_2 \times b_4)$

5. SAT Solver in $\Pi^{\eta\epsilon}$

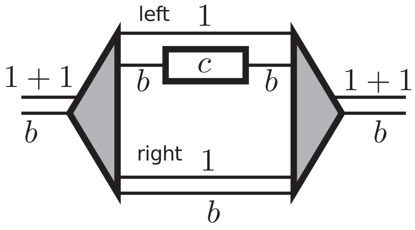
We illustrate the expressive power of first-class constraints represented by fractional types. To understand the intuition, recall the definition of $trace$ for the multiplicative fragment of $\Pi^{\eta\epsilon}$. Given $f : a \times c \leftrightarrow b \times c$, we have $trace^{\times} f : a \leftrightarrow b$:

$$trace^{\times} f = uniti \fatsemi (id \times \eta^{\times}) \fatsemi (f \times id) \fatsemi (id \times \epsilon^{\times}) \fatsemi unite$$

This circuit uses $\eta^{\times}$ to generate all possible c-values together with an associated $(1/c)$-constraint. It then applies f to the pair (a, c). The function f must produce an output (b, c') for each such input. If the input c and the output c' are the same they can be annihilated by $\epsilon^{\times}$; otherwise the execution gets stuck and this particular choice of c is pruned.

A large class of constraint satisfaction problems can be expressed using $trace^{\times}$. We illustrate the main ideas with the implementation of a SAT-solver. We proceed in small steps, reviewing some of the necessary constructions presented in our earlier chapters.

5.1. Booleans and Conditionals. Given any combinator $c : b \leftrightarrow b$ we can construct a combinator called if_c : *bool* $\times\, b \leftrightarrow$ *bool* $\times\, b$ in terms of c, where if_c behaves like a one-armed if-expression. If the supplied boolean is $true$ then the combinator c is used to transform the value of type b. If the boolean is $false$, then the value of type b remains unchanged. We can write down the combinator for if_c in terms of c as $distrib \fatsemi ((id \times c) + id) \fatsemi factor$.
The diagram below shows the input value of type $(1 + 1) \times b$ processed by the distribute operator $distrib$, which converts it into a value of type $(1 \times b) + (1 \times b)$. In the $left$ branch, which corresponds to the case when the boolean is $true$ (i.e. the value was $left$ ()), the combinator c is applied to the value of type b. The right branch which corresponds to the boolean being $false$ passes along the value of type b unchanged.

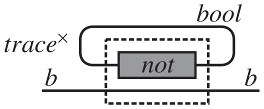


The combinator if_{not} has type *bool* $\times$ *bool* $\leftrightarrow$ *bool* $\times$ *bool* and negates its second argument if the first argument is $true$. This gate if_{not} is often referred to as the $cnot$ gate. An equivalent construction that is useful is $else_{not}$ where we negate the second argument only if the first is false.

Similarly, we can iterate the construction of if_c to check several bits. The gate if_{cnot}, which we may also write as if^2_{not}, checks two booleans and negates the result wire only if they are both true. The gate if^2_{not} is well known as the Toffoli gate and is a universal reversible gate. We can generalize this construction to if^n_{not} which checks n bits and negates the result wire only if they are all true.

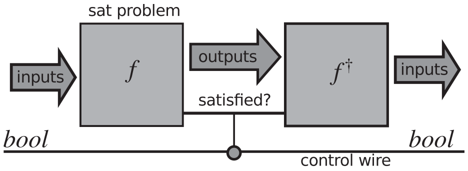
5.2. Cloning. Although cloning is generally not allowed in reversible languages, it is possible at the cost of having additional constant inputs. For example, consider the gate $else_{not}$. Generally, the gate maps (false, a) to $(\mathit{false}, \mathit{not}\ a)$ and (true, a) to (true, a). Focusing on the cases in which the second input is true, we get that the gate maps $(\mathit{false}, \mathit{true})$ to $(\mathit{false}, \mathit{false})$ and $(\mathit{true}, \mathit{true})$ to $(\mathit{true}, \mathit{true})$, i.e., the gate clones the first input. A circuit of n parallel $else_{not}$ gates can hence clone n bits. They also consume n true inputs in the process. Let us call this construction $clone^n_{bool}$.

5.3. Construction of the Solver. The key insight underlying the construction comes from the fact that we can build *annihilation circuits* such as the one below:



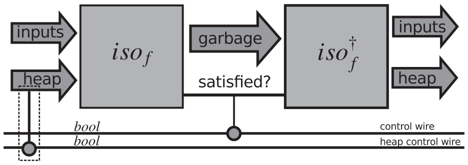
The circuit constructs a boolean b and its dual $1/b$, negates one of them and attempts to satisfy the constraint that they are equal which evidently fails.

With a little work, we can modify this circuit to only annihilate values that fail to satisfy the constraints represented by a SAT-instance f. In more detail, an instance of SAT is a function/circuit f that given some boolean inputs returns true or false which we interpret as whether the inputs satisfy the constraints imposed by the structure of f. Because we are in a reversible world, our instance of SAT must be expressed as an isomorphism: this is easily achieved as shown in Sec. 5.4 below. Assuming that f is expressed as an isomorphism, we have enough information to reconstruct the input from the output. This can be done by using the adjoint of f. At this point we have, the top half of the construction below:

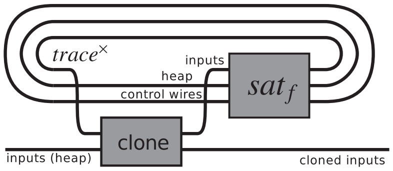


To summarize, the top half of the circuit is the identity function except that we have also managed to produce a boolean wire labeled satisfied? that tells us if the inputs satisfy the desired constraints. We can take this boolean value and use it to decide whether to negate the control wire or not. Thus, the circuit achieves the following goal: if the inputs do not satisfy f, the control wire is negated. We can now use $trace^\times$ to annihilate all these bad values because the control wire acts like the closed-loop not in the previous construction.

5.4. Final Details. Any boolean expression $f : \mathit{bool}^n \rightarrow \mathit{bool}$ can be compiled into the isomorphism $iso_f : \mathit{bool}^h \times \mathit{bool}^n \leftrightarrow \mathit{bool}^g \times \mathit{bool}$ where the extra bits bool^h and bool^g are considered as heap and garbage. Constructing such an isomorphism has been detailed before [James and Sabry, 2012a, Toffoli, 1980]. The important relation to note is that applying iso_f to some special heap values and an input *bs* produces some bits that can be ignored and the same output that f would have produced on bs. We can ensure that the heap has the appropriate initial values by checking the heap and negating a second control wire, if the values do not match, i.e., the dotted part in the diagram below.



Let us call the above construction which maps inputs, heap, and control wires to inputs, heap, and control wires as sat_f. The SAT-solver is is completed by tracing the sat_f and cloning the inputs using $clone^n_{\mathit{bool}}$.



When the solver is fed inputs initialized to $true$, it clones only those inputs to sat_f that satisfy f and the heap constraints. In the case of unique-SAT the solver will produce exactly 0 or 1 solutions. In the case of general SAT, the solver will produce solutions as determined by the semantics of the top-level interaction (see discussion in Sec. 3).

9
Discussion

Whether you can observe a thing or not depends on the theory which you use.
It is the theory which decides what can be observed.

Albert Einstein

(during Heisenberg's 1926 lecture, Berlin)

Starting with the notion that closed physical systems are the "purest," we have developed a pure language in which computations preserve information and proposed this language as a foundational one to which other languages can be translated to expose their implicit effects. We show that even what are often called "pure functional languages" exhibit computational effects related to the creation and deletion of information. This development re-asserts the fact that information is a significant computational resource that should, in many situations, be exposed to programmers and static analysis tools.

We believe that this thesis is only the beginning of this line of research and much work needs to be done to fully understand the connections of information preserving computation to semantics, logic and physics. Some of these broad themes are explored in some detail here. This chapter explores several related areas of research, highlighting related developments, possible applications and areas of further research.

1. Reversible Computing

Reversible computation is a well established area of research. The original motivation for research into reversible computation comes from the Landauer principle [Landauer, 1961]. The Landauer principle laid the foundation for the idea that circuits that don't lose information, i.e. ones that are logically reversible, will also not dissipate heat and thereby consume less energy. Thus reversible computation started as the quest for energy efficient computing. With the recent interest in quantum Computation, reversible computation is sometimes viewed as a special case of the broader goal of quantum computation (see also Sec. 4). Another immediate goal of reversible computation is to study *ad hoc* instances of reversible computations such as database transactions, mechanisms for data provenance, checkpoints, stack and exception traces, logs, backups, rollback recoveries, version control systems, reverse engineering, software transactional memories, continuations, backtracking search, and multiple-level "undo" features in commercial applications in a more principled fashion.

Reversible Logic. One of the seminal papers in this area is Toffoli [1980] which has been discussed in Ch. 7 Sec. 1. While much inspiration for this thesis was derived from Toffoli's work, important differences exist. Toffoli only deals with finite valued boolean circuits. In this setting he demonstrates that any irreversible function from m bits to n bits can be embedded into a reversible function from $m + h$ bits to $n + g$ bits. We extend Toffoli's framework to deal with arbitrary algebraic data-types, i.e. not just booleans, and deal with infinite-valued recursively defined data types such as nat. Further, Π programs[1] have a term-language in which they can be expressed, instead of working directly with truth tables which require manually checking that the required reversibility conditions hold.

Conservative Logic. The idea that physical conservation laws might be computationally relevant appears in Fredkin and Toffoli [1982] as *conservative logic* (see also Zuliani [2001]). Conservative logic focused on reversible boolean circuits wherein the number of set bits, i.e. the number of 1-states are conserved in the process of computation. The idea being that 1-states represent the presence of energy (high voltage state) and this energy should be conserved. While this is similar in spirit to the conservation of information in Π, it represents a low-level commitment to how energy and information is encoded and transmitted in a specific implementation choice for boolean circuits. In contrast, Π deals directly with information content independent of its specific realization in hardware.

Aside from the boolean circuit based models, reversible computation has been studied in the context of high-level languages (both imperative and functional) [Baker, 1992b, Matos, 2003, Mu et al., 2004, Yokoyama et al., 2008], low-level languages [Stoddart, 2003], cellular automata [Morita, 2001], primitive computational models like Turing machines and λ-calculus, abstract machines, term rewrite systems [Abramsky, 2005b] and logical foundations such as GoI [Mackie, 2011]. Our pure language is logically reversible and hence shares some common features with previously developed reversible languages although none are based on the simple notion of isomorphisms between types. We briefly discuss some of these.

[1] In many sections we have taken the liberty of referring to just Π even though the specific discussion may also be relevant to Π^o and their effectful meta-languages. These should however be apparent from their context.

- Janus is a high-level sequential imperative language for reversible computation developed by Lutz and Derby [1982] and Yokoyama and Glück [2007]. Reversible destructive updates have appeared in other languages as well, such as psiLisp of Baker [1992a]. Janus programs can be inverted by executing program statements in reverse order. Conditionals such as if require reverse execution of the program to follow the branch that would have been taken during forward execution. This is accomplished by having the programmer specify 'exit predicates' on join-points in the program. During reverse execution these exit predicates must determine which branch would have been taken during forward execution of the program sequence. Thus they serve as the dual of the branch predicates. However these exit predicates are application specific, are not easily derived from the branch predicates and require the programmer to manually derive them based on their insight into the specific application. This differs in essence from Π in that all well-typed programs are reversible by construction, independent of programmer discipline.
- Several models of reversible computation such as reversible Turing machines [Axelsen and Glück, 2011, Bennett, 1973, Lecerf, 1963], the reversible SECD machine [Kluge, 2000] and the reversible λ-calculus [Grattage, 2006, van Tonder, 2004] start with essentially irreversible models of computation and *add reversibility*. This added reversibility takes one of several forms – either the irreversible programs are compiled into reversible ones wherein the compilation process adds history tracking or other additional information to make the programs reversible, or programs are restricted from doing actions that would have been allowed in the original models. As an example of the later see Morita and Yamaguchi [2007], where the formalism restricts transitions in a Turing machine to achieve reversibility.
- Some models are inherently reversible by design, similar to Π. Notable ones are the stateful circuits based on the reversible rotatory element of Morita [2001] and the intermediate language of bi-orthogonal automata introduced by Abramsky [2005b].

- Applications of reversibility have also been studied in the context of concurrency [Danos et al., 2007] and several reversible models of computation provide concurrency in a reversible setting and are reversible extensions of process calculi [Danos and Krivine, 2004, Krivine, 2012, Phillips et al., 2012]. This thesis does not study concurrency in the setting of Π. This is a topic for further research and some preliminary notes are added to Sec. 5.2.

The establishment of Π adds another dimension to the study of reversible computation – it formalizes the notion that irreversibility is a computational effect caused by the changes to the information content of programs and this effect can be structured by a type-and-effect system. A possible application of our work is that, in principle, such reversible programs could be automatically derived from common irreversible programs (which are typically easier to write) by translations similar to the one we presented. Summarizing:

- Π is a reversible by design and it does not presuppose an irreversible computational model such as Turing machines or λ-calculus.
- Reversibility is not achieved by tracking undo-history of the computation (such as with the reversible SECD machine).
- Executing reversible computation does not accumulate any global state in the circuit model (such as with Morita's reversible switching element or Mackie's GoI machines).
- Forward and backward execution have equal status in Π and execution in one direction is not any more special than execution in the other direction. Computations are partial isomorphisms and not just injective functions.
- Reversibility does not rely on the construction of any application specific predicates at branch and join points (such as with Janus).
- One can reason about program with a richer type system than by working at the level of bits.
- It captures the the gap between reversibility and irreversibility by a type-and-effect system.

We conclude this section with a note on perspective: We believe that reversibility is the right default for computation and that irreversibility is derived concept. The reason that this is not the conventional view today is because the right primitive calculus for reversible computation had not been discovered. Until now it was unclear that the essence of computation is already present at a very primitive level – we have shown that to even discuss equality of sets is to talk about computation. In other words, even our most primitive notions of equality, namely isomorphisms, already have computational content. To talk about equalities is to already talk about computation.

2. Information, Entropy and Privacy

The immediate reaction to the Landauer principle is to consider building reversible computing devices that never erase any information and hence — in the theoretical limit — consume no energy [Frank, 1999]. Our main interest however is of a more semantic nature: the energy dissipation associated with a computation clearly constitutes some kind of "side effect" in the sense of being an unintended or unforeseen consequence of the computation. This semantic link between irreversibility and computational effects opens the door to treating "information" as computational resource that should, in many situations, be exposed to programmers and static analysis tools.

This class of applications includes any application in which "information" plays a significant role, such as information flow security [Sabelfeld and Myers, 2003], energy-aware computing [Ma et al., 2008, Zeng et al., 2002], VLSI design [Macii and Poncino, 1996], and biochemical models of computations [Cardelli and Zavattaro, 2008]. Each of these domains recognizes that *information is physical*, i.e., that information is encoded, processed, and transmitted by physical means, and hence that *information manipulations have observable effects* such as leaking sensitive data, increasing energy consumption, increasing the area used by a circuit design, preventing backtracking, or slowing the speed of a chemical reaction. We now briefly explore the application of such a framework to security, privacy, circuit design and confidentiality.

2.1. Quantitative Information Flow. Research in the domain of quantitative information-flow security is aimed at tracking the amount of 'information leakage' through a computation [Clark et al., 2007, Malacaria, 2007, Massey, 1994]. Instead of devising ad hoc analyses, one can in our framework simply observe the program's types and see how much information has been erased.

As an example, consider a password checker, $check : bool \times bool \rightarrow bool$, which verifies if a 2-bit password is correct. The password checker takes a 2-bit user input and has access to a 2-bit secret password implemented as: $check\ (l_1, l_2) = (l_1 == 1)\ \&\&\ (l_2 == 1)$, i.e. the password is 11. The type $bool \times bool$ has 4 inhabitants and with probability $1/4$ an attacker guesses the real password ($\log 4$ bits learned) and with probability $3/4$ guesses wrong thus eliminating one candidate password ($\log 4 - \log 3$ bits learned), making the average information gained $1/4 \log 4 + 3/4(\log 4 - \log 3) = 0.8$. The password checker has thus hidden 1.2 bits of information from the attacker.

The Landauer principle implies that the 1.2 bits dissipated by $check$ must be accounted for as a logically irreversible erasure of 1.2 bits. The most optimum Π^o program that implements $check$ must `erase` the 1.2 bits. However since we erase only integral types in Π^o, an optimal implementation of $check$ must erase 2 bits at a minimum, which would manifest itself by a use of `erase` at type $bool \times bool$. Further, a finer analysis allowing us to capture the exact erasure of fractional bits, may be achieved by enriching the type system of ML_{Π^o} to track the probability of disjunctive branches, such as in the probabilistic λ-calculus [Pierro et al., 2005]. The fact that ML_{Π} and ML_{Π^o} types indicate a lower bound on a program's intrinsic secrecy makes it an interesting setting to study quantitative information flow security.

2.2. Orthogonality and Differential Privacy. Physical processes operate on physical representations of values which exist in space and have associated costs (e.g. amount of energy). Physical processes must not only be reversible but they must do so in a way that respects this additional structure. Fredkin and Toffoli [1982] captured this additional restriction on physical processes using what they called *conservative logic* in which values can only be shuffled around by computation, as discussed before (see Section 1). In quantum mechanics, this additional restriction is modeled in a different way using the mathematical structure of Hilbert spaces. Specifically, the fundamental building blocks of quantum computation are that (i) quantum states are equipped with an inner product that induces a norm (i.e., a metric), and that (ii) quantum transformations must be *unitary*, i.e., must be reversible transformations that preserve the norm induced by the inner product. The key definition is the following:

Definition 2.1 (Isometry). *Given a metric d_τ on values of type τ, a function $f : \tau_1 \rightarrow \tau_2$ is said to be an isometry iff $d_{\tau_2}(f(x), f(y)) = d_{\tau_1}(x, y)$ for all $x, y \in \tau_1$.*

In an apparently unrelated development, Reed and Pierce [2010] introduce a calculus for differential privacy. The fundamental building blocks of their calculus are (i) that types are equipped with a metric that defines what it means for values to be "close," and that (ii) functions must preserve distances between the values. The key definition is the following:

Definition 2.2 (c-sensitive function). *Given a metric d_τ on values of type τ, a function f : $\tau_1 \rightarrow \tau_2$ is said to be c-*sensitive *iff $d_{\tau_2}(f(x), f(y)) \leq c \cdot d_{\tau_1}(x, y)$ for all $x, y \in \tau_1$.*

In other words, the distance between the values x and y in the domain of the function cannot be magnified by more than a factor c. Clearly quantum evolution is restricted to the special case of 1-sensitive functions.

To further investigate this connection would require us to extend our model by associating a metric with each type. However, unlike previous work, the introduction of the metric in our case would be justified by additional physical considerations.

2.3. Entropy Dissipation of Circuits. Computing the entropy of digital circuits plays an important role in VLSI design. Computing entropy dissipation of circuits is useful for power estimation of logic circuits at the gate-transfer level, estimating heat dissipation of circuits and time delay of logic networks [Ma et al., 2008, Macii and Poncino, 1996]. However logic circuits expressed as logic gates or truth tables do not yield a direct insight into where information is created or deleted in circuits. Since ML_Π provides a setting in which the logical content of a circuits and information effects can be expressed as syntactically distinct operations, i.e. by making information effects explicit, it may aid entropy analysis of circuits. The development of information preserving logic gates, such as in Sec. 4, are a step in this direction.

2.4. Thermodynamic Cost of Confidentiality. Malacaria and Smeraldi [2012] establish the connection between secure computation and the 2nd law of thermodynamics. This work clarifies an important connection which has been suspected before [Clark et al., 2007, James and Sabry, 2012a, Landauer, 1961] – it connects the information erased by a computation, the confidentiality guaranteed by the program and the thermodynamic action (energy dissipation) of the program. As Malacaria and Smeraldi note:

> [..] dissipation, while of great practical importance, seems to have little foundational status in computer science.
>
> Here we established a fundamental relation between dissipation and secure computation by proving that two of the main metrics of confidentiality in computer security, namely information leakage and vulnerability, are essentially measures of dissipation in the thermodynamic sense. These results provide thermodynamic foundations for confidentiality, with Landauer's principle thus implying a fundamental lower bound to the energetic cost of secure computation.

Malacaria and Smeraldi show that we can look at the abstract process of computation as a thermodynamic process whose interactions with the environment are information effects. In a sense, Malacaria and Smeraldi have clarified the thermodynamic interpretation of information effects and its connection to information theory. This sets the stage for important foundational investigation into quantitative information flow security and the thermodynamic nature of computation.

3. Duality of Computation

In our usual interaction with computation we typically deal with values and functions – continuations are encountered less often and typically are second class citizens. However, research into continuations indicates that the world of values and functions is symmetric and dual to the world of continuations and delimited continuations. This deep symmetry was first suggested by Filinski [1989] and later explored by Curien and Herbelin [2000], Wadler [2003] and others [Curien and Munch-Maccagnoni, 2010, Munch-Maccagnoni, 2009, Zeilberger, 2010]. In the spirit of Curien and Herbelin [2000], we use the phrase *duality of computation* to stand for this emerging picture of a deep symmetry in computation that comes from the study of continuations, evaluation order, computational effects, proof normalization, cut elimination strategies, semantics of linear logic and double negation transformations.

This section outlines the fragment of this story that is relevant to $\Pi^{\eta\epsilon}$ and its negative and fractional types. The idea of negative types has appeared many times in the literature and has often been related to some form of continuations. Fractional types are less common but have also appeared in relation to parsing natural languages.

Declarative Continuations. In his Masters thesis, Filinski [1989] proposes that continuations are a *declarative* concept. He introduces a symmetric extension of the λ-calculus in which call-by-value is dual to call-by-name and values are dual to continuations. The symmetric calculus contains a “value” fragment and a “continuation” fragment which are mirror images. Pairs and sums are treated as duals in the sense that the “value” fragment includes pairs whose mirror image in the “continuation” fragment are sums. Furthermore, Filinski [1989] shows that functions are dual to delimited continuations.

In more detail, a value (v_1, v_2) can be consumed by either applying a continuation that takes its first component, i.e. fst, or a continuation that takes the second component, i.e. snd. In this sense, if the value is a pair, the continuation has a choice. Thus the dual of the value type $t_1 \times t_2$ is the continuation type $\neg t_1 + \neg t_2$. Similarly for value types $t_1 + t_2$, the continuation is forced to consume either t_1 or t_2 depending on the value being $left\ v_1$

or $right\ v_2$, and has no choice. Thus the dual of the value type $t_1 + t_2$ is the continuation type $\neg t_1 \times \neg t_2$. In other words, values and continuations are related by a De Morgan style duality.

One way to describe $\Pi^{\eta\epsilon}$, is that $\Pi^{\eta\epsilon}$ extends Π with functions. In the spirit of Filinski [1989], we believe that values and continuations, functions and delimited continuations coexist and in a setting which handles the creation and the consumption of values carefully having one automatically implies having the other. And indeed, we automatically get its dual form of $\Pi^{\eta\epsilon}$ functions as explained in Chapter 8. However, there are important differences between the dualities in $\Pi^{\eta\epsilon}$ and those in Filinski's calculus.

The duality obtained in $\Pi^{\eta\epsilon}$ is very different from the dualizing '$\neg$' operation. In $\Pi^{\eta\epsilon}$ we have two distinct axes of duality, negation and division. The deep reason underlying this difference needs much more study. One possible reason for the difference is information preservation and how it relates to homotopy cardinality/Euler characteristic of $\Pi^{\eta\epsilon}$ [Leinster, 2008, Loeb, 1992, Schanuel, 1991]. In other words, it may be the case that the De Morgan style '$\neg$' is collapsing negation and fractionals into one dualizing operation by introducing an information effect which meshes additive and multiplicative components together. This is reminiscent of linear logic where we do get notions such as "additive pairs" and "multiplicative sums". Can the previously observed duality between values and continuations can be teased into two dualities: a duality between values flowing in one direction or the other and a duality between aggregate values composing and decomposing into smaller values? Arguably each of the dualities is more natural than a duality that maps regular values to a conflated notion of negative and fractional types requiring notions like "additive pairs" and "multiplicative sums." The treatment of De Morgan duality as an information effect and its connection to the Euler characteristic of $\Pi^{\eta\epsilon}$ deserve further study. On a related note, see Sec. 3 of Baez and Dolan [2000] for a discussion on the categorification of negative numbers.

The rest of this section touches upon other relevant developments in this space.

Duality of Computation. The duality between call-by-name and call-by-value was further investigated by Selinger [2001] using Control Categories. Curien and Herbelin [2000] also introduce a calculus that exhibits symmetries between values and continuations and between call-by-value and call-by-name. The calculus includes the type $A - B$ which is the dual of implication, i.e., a value of type $A - B$ is a context expecting a function of type $A \to B$. Alternatively a value of type $A - B$ is also explained as a *pair* consisting of a value of type A and a continuation of type B. This is to be contrasted with our interpretation of a value of that type as *either* a value of A or a demand for a value of type B. This calculus was further analyzed and extended by Wadler [2003, 2005]. The extension gives no interpretation to the subtraction connective and like the original symmetric calculus of Filinski, introduces a duality that relates sums to products and vice-versa.

Subtractive Logic. Rauszer [1974a,b, 1980] introduced a logic which contains a dual to implication. Her work has been distilled in the form of *subtractive logic* [Crolard, 2001] which has recently been related to coroutines [Crolard, 2004] and delimited continuations [Ariola et al., 2009]. In more detail, Crolard explains the type $A - B$ as the type of *coroutines* with a local environment of type A and a continuation of type B. The description is complicated by what is essentially the desire to enforce linearity constraints so that coroutines cannot access the local environment of other coroutines.

Negation in Classical Linear Logic. Filinski [1992] uses the negative types of linear logic to model continuations. Reddy [1991] generalizes this idea by interpreting the negative types of linear logic as *acceptors*, which are like continuations in the sense that they take an input and return no output. Acceptors however are also similar in flavor to logic variables: they can be created and instantiated later once their context of use is determined. Although a formal connection is lacking, it is clear that, at an intuitive level, acceptors are entities that combine elements of our negative and fractional types.

The Lambek-Grishin Calculus. The “parsing-as-deduction” style of linguistic analysis uses the Lambek-Grishin calculus with the following types: product, left division, right division, sum, right difference, and left difference [Bernardi and Moortgat, 2010]. The division and difference types are similar to our types but because the calculus lacks commutativity and associativity and only has limited notions of distributivity, each connective needs a left and right version. The Lambek-Grishin calculus exhibits two notions of symmetry but they are unrelated to our notions. In particular, the first notion of symmetry expresses commutativity and the second relates products to sums and divisions to subtractions. In contrast, our two symmetries relate sums to subtractions and products to divisions.

We conclude with some thoughts:

- Negative and fractional types have an elementary and familiar interpretation borrowed from the algebra of rational numbers. One can write any algebraic identity that is valid for the rational numbers and interpret it as an isomorphism with a clear computational interpretation: negative values flow backwards and fractional values represent constraints on the context. None of the systems above has such a natural interpretation of negative and fractional types.
- Because we are not in the context of the full λ-calculus, which allows arbitrary duplication and erasure of information, values of negative and fractional types are first-class values that can flow anywhere. The information-preserving computational infrastructure guarantees that, in a complete program, every negative demand will be satisfied exactly once, and every constraint imposed by a fractional value will also be satisfied exactly once. This property is partly shared with systems that are based on linear logic; other systems must impose ad hoc constraints to ensure negative and fractional values are used exactly once.

3.1. Linear Logic, Geometry of Interaction (GoI) and Higher-Order Functions. Linear logic was introduced in Girard [1987] as a logic that had a physical flavor and a notion of conservation of "truth" resources. Linear logic is also sometimes described as an intermediate logic between classical and intuitionistic logics, having a multiplicative and additive fragments and a De Morgan style dualizing '$\neg$' that transports one between these fragments.

Linearity however must not be confused with information preservation. Consider the function $f(x) = \mathit{if}\ x\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{true}$ which is extensionally equivalent to the constant function $f'(x) = \mathit{true}$. In a linear type system that tracks the *syntactic* occurrences of variables, f would be deemed acceptable because x is linearly used. However function f' is not information-preserving.

Geometry of Interaction (GoI) was developed by Girard [1989] as part of the development of linear logic. It was given a computational interpretation by Abramsky and Jagadeesan [1994] and also Abramsky et al. [2002], and was developed into a reversible model of computing by Mackie [1995, 2011]. GoI is characterized by its computational model which expresses computation as the flow of values (particles) along wires in a manner similar to Π's wiring diagrams. Preliminary investigations suggest that many of the GoI machine constructions [Mackie, 2011] can be simulated in $\Pi^{\eta\epsilon}$ by treating Mackie's bi-directional wires as pairs of wires in $\Pi^{\eta\epsilon}$ and replacing the machine's global state with a typed value on the wire that captures the appropriate state. This connection is exciting because when viewed through a Curry-Howard lens it suggests that the logical interpretation of $\Pi^{\eta\epsilon}$ would be a linear-like logic with a notion of resource preservation and with a natural computational interpretation.

However, the exact relationship between GoI and Π wiring diagrams is unknown. GoI for the additive fragment of linear logic is not very well developed and this may in part because of its handling of resources. In the rules below we see that linear logic does allow for dropping alternatives:

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \mathit{left\ plus} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \mathit{right\ plus}$$

In the linear logic interpretation of the above, the disjunction presents the choice of having one of the resources or the other – i.e. a classical or – and in this sense, resources are preserved. Indeed, in this setting we can view linear logic as removing projections while retaining injections – i.e. linear logic has refined its structure one step by removing information effects from its multiplicative fragment. In contrast, Π disjunctions cannot be injected into without an information effect.

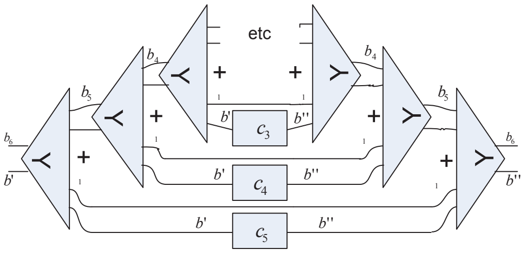
More generally, one view of a logic is that it is a discourse language to desribe formally the "truth" of some model. Logics describe the truth or falsehood of their model. Despite the appealing characteristics of linear logic it is unclear what it talks about becuase it has no natural models and no clear computational interpretation. Simply put, it is not clear what linear logic talks about. This is probably the most foundational difference between linear logic and the Π family. With Π (and $\Pi^{\eta\epsilon}$), the logical content of the type systems is very familiar – they correspond to the whole numbers (and the rational numbers respectively). Similarly the computational and categorical interpretation of these systems are also well understood. As noted in Section 3 understanding the role of De Morgan style dualizing operation and the exact connection between our wiring diagrams and GoI might help explain the gap between linear logic and Π.

3.2. Other approaches to Higher-Order Functions. Finally, we conclude with a note on the construction of higher order functions. One way to construct higher order functions is to have η and ϵ primitives as we did with $\Pi^{\eta\epsilon}$. However it is well known that these may also be constructed, instead of being taken as primitive.

Int construction. The Int construction [Joyal et al., 1996] (and the closely related G construction [Abramsky and Jagadeesan, 1994]) are known as a means of constructing a compact closed category from a given traced monoidal category. Could we have derived a structure similar to $\Pi^{\eta\epsilon}$ by applying the Int construction to Π^{o} – which is a traced monoidal category? In principle this is indeed possible, however doing the Int construction over the additive traced monoid does not preserve the multiplicative monoid. In particular, in the resulting category rules such as distributivity do not hold between the lifted additive and multiplicative fragments.

Encoding functions over finite-types. In the case of Π, i.e. in the world of finite types, there is simpler approach. The function space between two finite types is itself a finite type. Precisely, the total number of isomorphisms, $c_0 ... \; c_{n-1}$, of type $b' \leftrightarrow b''$, is given by the factorial of the number of inhabitants of the type b' (we write $b'!$). Thus, to denote higher-order functions, we can use a type, $b_{b' \leftrightarrow b''}$, which has the form $1 + 1 + 1 + ...$ with precisely $b'!$ number of 1s. Each of the values of $b_{b' \leftrightarrow b''}$ denote a $b' \leftrightarrow b''$ isomorphism.

Given this encoding of higher order functions, function application works as follows: at each application site of $b' \leftrightarrow b''$ to b', we distribute input argument b' over the function encoding, $b_{b' \leftrightarrow b''}$, thereby applying the appropriate isomorphism c_i as shown in the diagram below. The diagram shows the case of the function space $1 + (1 + 1) \leftrightarrow (1 + 1) + 1$ where b' has 3 inhabitants and $b'! = 6$.



The above encoding of higher order types however is only applicable for finite types. The strong similarities between Π^o and linear logic, suggest that the idea used by Mackie in the context of the geometry of interaction [Mackie, 1995, 2011] might be applicable in our setting.

4. Quantum Computing

One understanding of quantum computing is that it exploits the laws of physics to build faster machines (perhaps). Another more foundational understanding is that it provides a computational interpretation of physics, and in particular directly addresses the question of interpretation of quantum mechanics.

From a programming perspective, many programming models of quantum computing start with the λ-calculus as the underlying classical language and add quantum features on top of it [Altenkirch and Grattage, 2005, Duncan, 2006, Selinger and Valiron, 2006, van Tonder, 2004]. This strategy is natural given that the λ-calculus is the canonical classical computational model. However, since quantum evolution is reversible, this strategy complicates the development of quantum languages as it forces the languages to devise complicated ways to restrict the implicit duplication and erasure of information in the classical sublanguage. A simpler strategy that loses no generality is to build the quantum features on top of a reversible classical language: this keeps the language simple while still enabling the λ-calculus constructs to be encoded using explicit operators for erasure and duplication.

In a little known document, Rozas [1987] uses continuations to implement the transactional interpretation of quantum mechanics [Cramer, 1986] which includes as its main ingredient a fixpoint calculation between waves or particles traveling forwards and backwards in time. The idea of entanglement, that an action on one particle is “instantaneously” communicated to the other, is analogous to how unifying one value affects its dual pair which is possibly in another part of the computation. While our model sheds no light on whether this is related to how nature computes but it is again interesting that we can directly realize the idea using the primitives of $\Pi^{\eta\epsilon}$.

The works of Bob Coecke and Samson Abramsky (and their group, called the Quantum Foundations group at Oxford University), Peter Selinger, John Baez and others stem from a desire to clarify the mathematical foundations used to describe quantum physics. In doing so they have moved away from using Hilbert spaces and partial differential equations as their primary language to categorical structure and normalization rules of these structures as a way of describing quantum states and quantum evolution [Abramsky and Coecke, 2004, 2008, 2009, Baez and Dolan, 2000, Coecke and Duncan, 2008, Selinger, 2004, 2007]. The primary descriptive structure here is dagger traced symmetric monoidal categories and variations on this, i.e. by restricting commutativity in case, adding η and ϵ etc. One notable distinction is that in the work of Abramsky and Coecke [2004, 2008], Coecke [2005] and much of their other work the focus has been on multiplicative structure, corresponding to tensor product, in Hilbert spaces – while the additive structure, corresponding to direct sum, has not been explored very much. The close similarity between the categorical structures of QM and $\Pi^{\eta\epsilon}$ and their string/wiring diagrams suggests that a strong connection between quantum physics and computation may exist.

While the existence of the computational model Π, Π^{o} and $\Pi^{\eta\epsilon}$ do not provide any evidence on how Nature works – it is a compelling coincidence that the study of type isomorphisms and their computational model, with a desire to clarify the meaning of computational effects, leads to similar models that research into clarifying the mathematical foundations of Quantum Physics have led to.

5. Other Future Work

5.1. Computing with Algebraic Numbers. Algebraically, the move from Π to $\Pi^{\eta\epsilon}$ corresponds to a move from a ring-like structure to a full field/meadow. Our language $\Pi^{\eta\epsilon}$ captures the structure of one particular field: that of the rational numbers. As we have seen, computation in this field is quite expressive and interesting and yet, it has two fundamental limitations. First it cannot express general recursive datatypes, and second it does not include any specific datatype definitions. We believe these to be two orthogonal extensions: recursive types were considered in Chapter 4; arbitrary datatypes are however more exciting that plain rationals as each datatype definition can be viewed as a polynomial extension of the field. This essentially means that we start computing in the field of algebraic numbers, which includes square roots and imaginary numbers.

As crazy as it might seem, the type $\sqrt{2}$ and even the type $(1/2) + i(\sqrt{3}/2)$ "make sense." In fact the latter type is the solution to the polynomial $x^2 - x + 1 = 0$ which if re-arranged looks like $x = 1 + x \times x$ and perhaps more familiarly as the datatype of binary trees $\mu x.(1 + x \times x)$, i.e. $\mathit{tree} = \mathit{Leaf} \mid \mathit{Node}\ \mathit{tree}\ \mathit{tree}$. These types happen to have been studied extensively following a paper by Blass [1995] which used the above datatype of trees to infer an isomorphism between seven binary trees and one!

We can extended $\Pi^{\eta\epsilon}$ with the datatype declaration for binary trees and build a witness for this isomorphism that work as expected. However, not every combinator constructed from algebraic manipulation is computationally meaningful. The isomorphism presented in [Fiore, 2004] for instance can be understood as taking any 7-tuple of binary trees as input and outputing one binary tree (and it dual map). In contrast, consider the following algebraically valid proof of the isomorphism in question:

$$\begin{array}{rcl} x^3 & = & x^2x \;=\; (x-1)x \;=\; x^2 - x \;=\; -1 \\ x^6 & = & 1 \\ x^7 & = & x^6x \;=\; x \end{array}$$

While the algebraic manipulation makes sense, the intermediate step of asking for an isomorphism between x^6 and 1 has no computational context – i.e. it's always the infinite loop.

In the setting of $\Pi^{\eta\epsilon}$ it means that the combinator corresponding to the above algebraic manipulation can be constructed but it diverges on all inputs (in both ways). This suggests that, in the field of algebraic numbers, some algebraic manipulations are somehow "more constructive" than others.

A related issue is that not all meaningful recursive types are meaningful polynomials. For instance $nat = \mu x.(1 + x)$ implies the polynomial $x = 1 + x$ which has no algebraic solutions without appeal to more complex structures with limits etc. That said, is computation possible in polynomial field extensions of the rationals possible? This is a fascinating open problem.

5.2. Hardware and Low-level languages. Recent interest in programming language based security [Bershad et al., 1995], security preserving compilation [Ahmed and Blume, 2008, 2011, Necula and Lee, 1998] and typed assembly languages [Morrisett et al., 1998, Necula, 1997] have revived interest in research into low-level languages, typically assembly language of some form, with better type systems and security guarantees.

In this context we consider the possibility of Π-based hardware. Most low-level languages today start with the assumption of some sort Von Neumann architecture and then try to add types, parallelism, security guarantees and more on top of this. However, what makes a low-level language a low-level language? One common answer is that the operations exposed by the language closely resemble the capabilities of the hardware. Modern hardware is concerned with shuffling values between registers, memory and register based operations and consequently our assembly languages reflects this. Can we have a fresh start?

Could our wiring diagrams be directly realized as hardware circuits? If so, this could pave the way for speculative execution and backtracking infrastructure in CPUs that are inherently reversible and use minimal book-keeping resources. In other words, can we treat Π as a low-level langauge?

Π can be considered a low-level language in two senses: First, it is low-level in a semantic sense – every primitive operation in Π does something that is semantically very simple and easy to specify. Second, Π is low-level in a hardware sense: most "gates" that comprise Π basically do the following - they either shuffle values around (i.e. they specify wiring), or they are concerned with the reversible creation/erasure of constants (have voltage sources and sinks), distribution and factoring (which are multiplexers and demultiplexers). The $trace$ operators are essentially feedback loops.

This idea can be extended to types. Types labeling wires should be thought of as a measure of the total information carrying capacity of the wire, as a fidelity measure of the wire. Since we never have wires with an infinite fidelity, full recursive types which have infinite values cannot be realized directly in hardware. Doing so would require a wire to carry a infinite number of distinguishable states. Instead the resolution available for the wires would correspond to fixed precision recursive types, as is the case with types such as `int16` and `int32` today. Furthermore:

- Π has builtin parallelism. Circuits c_1 and c_2 when laid out in parallel as $c_1 \otimes c_2$ can be evaluated in parallel by the hardware as these branches are completely independent of each other. Thus the language already contains a form of synchronization free parallelism built in.
- Π already has a strong notion of types intrinsic to its design. One does not need to retrofit a conventional assembly language with types, separation logic etc. This is important in the context of equivalence-preserving compilation and proof carrying code, where the target assembly language is equipped with rich type structure that permits invariants of high-level source languages to be captured.
- Much work has been done in devising reversible boolean circuits based on reversible primitive gates such as the *cnot* (controlled not) gate and the *toffoli* gate. Optimizing the layout of such circuits have been studied as well as means of optimizing heap and garbage. Many of these ideas could carry over to Π circuits as well.

- Π-based hardware is logically reversible and can achieve the goals of reduced power consumption and reduction of heat dissipation/energy as promised by research into reversible computation.
- Finally, and perhaps most importantly, we can reason about information effects at the lowest level of our computation. This has implications for security sensitive code, interaction of multiple programs running on the same hardware etc.

6. Conclusion

We believe that the difficulties in designing a computational model that is expressible at a quantum mechanical level and the difficulties of postulating a unified view of computational effects in programming languages stem from the same source. These difficulties stem from the ad hoc handling of the information content of computations in our usual computing models.

The study of programming languages, their typing, their semantics, their computational effects, and even their security properties can tremendously benefit from taking the physical aspect of computation seriously – information is physical and its transformation and disspiation has both conceptual and physical significance. To tackle this at a foundational level, we must appreciate that choices of formal equality or reduction rules that define our computational models have an inherent 'cost'. In this thesis we have argued that the most basic unit of that cost is information. Stated differently, formal equalities must take into account their information effects. Our work provides the conceptual foundation for such investigations.

Once the information content of computation is taken seriously, a computational model emerges that bridges the gap between the information theoretic analysis of computations (for example in the domain of quantitative information flow security) and the traditional world of type and effect systems.

Bibliography

Abramsky, S. (2005a). Abstract scalars, loops, and free traced and strongly compact closed categories. In *CALCO*.

Abramsky, S. (2005b). A structural approach to reversible computation. *Theor. Comput. Sci.*, 347:441–464.

Abramsky, S. (2009). No-cloning in categorical quantum mechanics. *arXiv preprint arXiv:0910.2401*.

Abramsky, S. and Coecke, B. (2004). A categorical semantics of quantum protocols. In *Logic in Computer Science*. IEEE.

Abramsky, S. and Coecke, B. (2008). Categorical quantum mechanics.

Abramsky, S. and Coecke, B. (2009). Abstract physical traces. *arXiv preprint arXiv:0910.3144*.

Abramsky, S., Haghverdi, E., and Scott, P. (2002). Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5).

Abramsky, S. and Jagadeesan, R. (1994). New foundations for the geometry of interaction. *Inf. Comput.*, 111:53–119.

Ahmed, A. and Blume, M. (2008). Typed closure conversion preserves observational equivalence. In *ICFP*, pages 157–168. ACM.

Ahmed, A. and Blume, M. (2011). An equivalence-preserving cps translation via multi-language semantics. *SIGPLAN Not.*, pages 431–444.

Altenkirch, T. and Grattage, J. (2005). A functional quantum programming language. In Panangaden, P., editor, *LICS*, pages 249–258. IEEE Computer Society Press.

Ariola, Z. M., Herbelin, H., and Sabry, A. (2009). A type-theoretic foundation of delimited continuations. *Higher Order Symbol. Comput.*, 22:233–273.

Axelsen, H. B. (2011). Clean translation of an imperative reversible programming language. In *CC*.

Axelsen, H. B. and Glück, R. (2011). What do reversible programs compute? In *FOSSACS/ETAPS*, pages 42–56. Springer-Verlag.

Baez, J. and Dolan, J. (2000). From finite sets to Feynman diagrams. *arXiv preprint math/0004133*.

Baez, J. and Stay, M. (2011). Physics, topology, logic and computation: a Rosetta Stone. *New Structures for Physics*, pages 95–172.

Baez, J. C. and Dolan, J. (1998). Categorification. In Higher Category Theory, eds. Ezra Getzler and Mikhail Kapranov, Contemp. Math. 230, American Mathematical Society, Providence, Rhode Island, 1998, pp. 1-36.

Baker, H. G. (1992a). Lively linear Lisp: — look ma, no garbage! —. *SIGPLAN Not.*, 27:89–98.

Baker, H. G. (1992b). Nreversal of fortune - the thermodynamics of garbage collection. In *Proceedings of the International Workshop on Memory Management*, pages 507–524. Springer-Verlag.

Barr, M. and Wells, C. (1995). *Category theory for computing science (2. ed.)*. International Series in Computer Science. Prentice Hall.

Bauer, A. and Pretnar, M. (2012). Programming with algebraic effects and handlers. *arXiv preprint arXiv:1203.1539*.

Bennett, C. (2003). Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 34(3):501–510.

Bennett, C. and Landauer, R. (1985). The fundamental physical limits of computation. *Scientific American*, 253(1):48–56.

Bennett, C. H. (1973). Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532.

Bergstra, J. A., Hirshfeld, Y., and Tucker, J. (2008). Fields, meadows and abstract data types. In *Pillars of Computer Science*, pages 166–178.

Bergstra, J. A., Hirshfeld, Y., and Tucker, J. V. (2009). Meadows and the equational specification of division. *Theor. Comput. Sci.*, 410(12-13):1261–1271.

Bernardi, R. and Moortgat, M. (2010). Continuation semantics for the Lambek–Grishin calculus. *Inf. Comput.*, 208:397–416.

Bershad, B., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., and Sirer, E. (1995). Protection is a software issue. In *Hot Topics in Operating Systems, 1995.(HotOS-V), Proceedings., Fifth Workshop on*, pages 62–65. IEEE.

Bérut, A., Arakelyan, A., Petrosyan, A., Ciliberto, S., Dillenschneider, R., and Lutz, E. (2012). Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483.

Blass, A. (1995). Seven trees in one. *Journal of Pure and Applied Algebra*, 103(1-21).

Bloom, S. and Ésik, Z. (1993). *Iteration theories: the equational logic of iterative processes*. Springer-Verlag.

Boole, G. (1854). *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*, volume 2. Walton and Maberly.

Bowman, W. J., James, R. P., and Sabry, A. (2011). Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Workshop on Reversible Computation*.

Cardelli, L. and Zavattaro, G. (2008). On the computational power of biochemistry. In *Third International Conference on Algebraic Biology*.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical report, Cornell Univ.

Clark, D., Hunt, S., and Malacaria, P. (2007). A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15:321–371.

Coecke, B. (2005). Kindergarten quantum mechanics. *arXiv preprint quant-ph/0510032*.

Coecke, B. and Duncan, R. (2008). Interacting quantum observables. *Automata, Languages and Programming*, pages 298–310.

Cramer, J. (1986). The transactional interpretation of quantum mechanics. *Reviews of Modern Physics*, 58:647–688.

Crane, L. and Yetter, D. N. (1996). Examples of categorification.

Crolard, T. (2001). Subtractive logic. *Theoretical Computer Science*, 254(1-2):151–185.

Crolard, T. (2004). A formulae-as-types interpretation of subtractive logic. *Journal of Logic and Computation*, 14(4):529–570.

Curien, P. and Munch-Maccagnoni, G. (2010). The duality of computation under focus. *Theoretical Computer Science*.

Curien, P.-L. and Herbelin, H. (2000). The duality of computation. In *ICFP*, New York, NY, USA. ACM.

Danos, V. and Krivine, J. (2004). Reversible communicating systems. *Concurrency Theory*, pages 292–307.

Danos, V., Krivine, J., and Tarissan, F. (2007). Self-assembling trees. *Electronic Notes in Theoretical Computer Science*.

Danvy, O. (2009). From reduction-based to reduction-free normalization. In *AFP*, pages 66–164. Springer-Verlag.

Deutsch, D. (1997). *The Fabric of Reality*. Penguin.

Di Pierro, A., Hankin, C., and Wiklicky, H. (2006). Reversible combinatory logic. *MSCS*, 16:621–637.

Duncan, R. (2006). *Types for quantum computation*. PhD thesis, Oxford University.

Dwork, C. (2006). Differential privacy. In *ICALP (2)'06*, pages 1–12.

Felleisen, M., Friedman, D., and Kohlbecker, E. (1987). A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237.

Felleisen, M. and Hieb, R. (1992a). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271.

Felleisen, M. and Hieb, R. (1992b). The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2):235–271.

Filinski, A. (1989). Declarative continuations: An investigation of duality in programming language semantics. In *Category Theory and Computer Science*, pages 224–249. Springer.

Filinski, A. (1992). Linear continuations. In *POPL*, pages 27–38. ACM Press.

Filinski, A. (1996). Controlling effects. Technical report, DTIC Document.

Fiore, M. (2004). Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM.

Frank, M. P. (1999). *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology.

Fredkin, E. and Toffoli, T. (1982). Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253.

Ghica, D. R. (2007). Geometry of synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375. ACM.

Girard, J. (1989). Geometry of interaction 1: Interpretation of System F. *Studies in Logic and the Foundations of Mathematics*.

Girard, J.-Y. (1987). Linear logic. *Theor. Comput. Sci.*, 50:1–102.

Glück, R. and Kawabe, M. (2005). Revisiting an automatic program inverter for Lisp. *SIGPLAN Not.*, 40:8–17.

Grattage, J. (2006). *QML: A functional quantum programming language*. PhD thesis, The University of Nottingham.

Halmos, P. (1960). *Naive Set Theory*. Van Nostrand. Reprinted by Springer-Verlag, Undergraduate Texts in Mathematics, 1974.

Hasegawa, M. (1997). Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*. Springer-Verlag.

Hasegawa, M. (2009). On traced monoidal closed categories. *MSCS*.

Hatcliff, J. and Danvy, O. (1994). A generic account of continuation-passing styles. In *POPL*, pages 458–471. ACM.

Higgins, P. (1971). *Notes on categories and groupoids*, volume 32. Van Nostrand Reinhold.

Huelsbergen, L. (1996). A logically reversible evaluator for the call-by-name lambda calculus. *InterJournal Complex Systems*, 46.

Hughes, J. (2000). Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111.

James, R. P. and Sabry, A. (2012a). Information effects. In *POPL*.

James, R. P. and Sabry, A. (2012b). Isomorphic Interpreters from Small-Step Abstract Machines. In *Reversible Computation*.

James, R. P. and Sabry, A. (2012c). The Two Dualities of Computation : Negative and Fractional Types. Technical report, Indiana University.

Joyal, A., Street, R., and Verity, D. (1996). Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*. Cambridge Univ Press.

Kluge, W. E. (2000). A reversible SE(M)CD machine. In *International Workshop on Implementation of Functional Languages*, pages 95–113. Springer-Verlag.

Krivine, J. (2012). A verification technique for reversible process algebra. *Reversible Computation, Lecture Notes in Computer Science*.

Landauer, R. (1961). Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191.

Lawvere, F. and Schanuel, S. (1997). *Conceptual mathematics: a first introduction to categories*. Cambridge University Press.

Lecerf, Y. (1963). Machines de Turing réversibles. *Comptes Rendus*, pages 2597–2600.

Leinster, T. (2008). The Euler characteristic of a category. *Doc. Math*, 13(21-49):119.

Lindley, S., Wadler, P., and Yallop, J. (2010). The arrow calculus. *Journal of Functional Programming*, 20(01):51–69.

Loeb, D. (1992). Sets with a negative number of elements. *Advances in Mathematics*, 91(1):64–74.

Lutz, C. and Derby, H. (1982). Janus: a time-reversible language. *Unpublished report, Ph. D. Dissertation, California Institute of Technology*.

Ma, X., Huang, J., and Lombardi, F. (2008). A model for computing and energy dissipation of molecular QCA devices and circuits. *J. Emerg. Technol. Comput. Syst.*, 3(4):1–30.

Mac Lane, S. (1971). *Categories for the Working Mathematician*, volume 5. Springer-Verlag. Graduate Texts in Mathematics.

Macii, E. and Poncino, M. (1996). Exact computation of the entropy of a logic circuit. In *Proceedings of the 6th Great Lakes Symposium on VLSI*, Washington, DC, USA. IEEE Computer Society.

Mackie, I. (1995). The geometry of interaction machine. In *POPL*, pages 198–208.

Mackie, I. (2011). Reversible higher-order computations. In *Workshop on Reversible Computation*.

Malacaria, P. (2007). Assessing security threats of looping constructs. In *POPL*, pages 225–235. ACM.

Malacaria, P. and Smeraldi, F. (2012). The thermodynamics of confidentiality. In *CSF*, pages 280–290.

Massey, J. L. (1994). Guessing and entropy. In *Proceedings of the IEEE International Symposium on Information Theory*, page 204.

Matos, A. B. (2003). Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290:2063–2074.

Mazur, B. (2008). When is one thing equal to some other thing? *Proof and Other Dilemmas: Mathematics and Philosophy, Mathematical Association of America, Washington, DC*.

Mcbride, C. and Paterson, R. (2007). Idioms : Applicative programming with effects. *Journal of Functional Programming*, 18.

Melliès, P. (2008). Categorical semantics of linear logic. *Panoramas et synthèses-Société mathématique de France*.

Melliès, P.-A. (2006). Functorial boxes in string diagrams. In *Proceedings of the 20th international conference on Computer Science Logic*, CSL'06. Springer-Verlag.

Minamide, Y., Morrisett, G., and Harper, R. (1996). Typed closure conversion. In *POPL*, pages 271–283, New York, NY, USA. ACM.

Moggi, E. (1989). Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, LICS, pages 14–23. IEEE.

Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93:55–92.

Morita, K. (2001). A simple universal logic element and cellular automata for reversible computing. In *Machines, Computations, and Universality*, Lecture Notes in Computer Science, pages 102–113. Springer Berlin Heidelberg.

Morita, K. and Yamaguchi, Y. (2007). A universal reversible Turing machine. In *Proceedings of the 5th international conference on Machines, computations, and universality*, MCU07, pages 90–98. Springer-Verlag.

Morrisett, G., Walker, D., Crary, K., and Glew, N. (1998). From system F to typed assembly language (extended version). In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 85–97.

Mu, S.-C., Hu, Z., and Takeichi, M. (2004). An injective language for reversible computation. In *MPC*, pages 289–313.

Munch-Maccagnoni, G. (2009). Focalisation and classical realisability. In *Computer Science Logic*. Springer.

Necula, G. and Lee, P. (1998). The design and implementation of a certifying compiler. In Cooper, K. D., editor, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344.

Necula, G. C. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris.

nLab (accessed on 19 April 2012). Evil. `http://ncatlab.org/nlab/show/evil`.

Penrose, R. (1971). Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*. Academic Press.

Phillips, I., Ulidowski, I., and Yuen, S. (2012). A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway. *Reversible Computation, Lecture Notes in Computer Science*.

Pierce, B. C. (1991). *Basic Category Theory for Computer Scientists*. MIT Press.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pierro, A. D., Hankin, C., and Wiklicky, H. (2005). Probabilistic lambda-calculus and quantitative program analysis. *J. Log. Comput.*, 15(2).

Pitt, D. H., Abramsky, S., Poigné, A., and Rydeheard, D. E., editors (1986). *Category Theory and Computer Programming, Tutorial and Workshop, Guildford, UK, September 16-20, 1985 Proceedings*, Lecture Notes in Computer Science. Springer.

Plotkin, G. (1977). LCF considered as a programming language. *Theoretical Computer Science*.

Plotkin, G. and Power, J. (2002). Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, pages 373–393. Springer.

Plotkin, G. and Power, J. (2004). Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163.

Plotkin, G. and Pretnar, M. (2009). Handlers of algebraic effects. *Programming Languages and Systems*.

Plotkin, G. D. (1975). Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159.

Quine, W. (1953). Reference and modality. *From a logical point of view*, 139:59.

Rauszer, C. (1974a). A formalization of the propositional calculus of H-B logic. *Studia Logica*, 33:23–34.

Rauszer, C. (1974b). Semi-boolean algebras and their applications to intuitionistic logic with dual operators. *Fundamenta Mathematicae*, 83:219–249.

Rauszer, C. (1980). An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathmaticae*, volume 167. Institut Mathématique de l'Académie Polonaise des Sciences.

Reddy, U. S. (1991). Acceptors as values: Functional programming in classical linear logic. Manuscript.

Reed, J. and Pierce, B. C. (2010). Distance makes the types grow stronger: a calculus for differential privacy. In *ICFP*, pages 157–168. ACM.

Rendel, T. and Ostermann, K. (2010). Invertible syntax descriptions: unifying parsing and pretty printing. In *Symposium on Haskell*, pages 1–12. ACM.

Ross, B. J. (1997). Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348.

Rozas, G. J. (1987). A computational model for observation in quantum mechanics. Technical report, MIT, Cambridge, MA, USA.

Sabelfeld, A. and Myers, A. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19.

Sabry, A. (1998). What is a purely functional language? *Journal of Functional Programming*, 8(01):1–22.

Sabry, A. (2008). Side effects. In *Wiley Encyclopedia of Computer Science and Engineering*. Wiley.

Sabry, A. and Wadler, P. (1997). A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941.

Schanuel, S. (1991). Negative sets have Euler characteristic and dimension. In *Category theory*, pages 379–385. Springer.

Selinger, P. (2001). Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical. Structures in Comp. Sci.*, 11:207–260.

Selinger, P. (2004). Towards a quantum programming language. *Mathematical Structures in Computer Science*.

Selinger, P. (2007). Dagger compact closed categories and completely positive maps. *Electronic Notes in Theoretical Computer Science*, 170:139–163.

Selinger, P. (2011). A survey of graphical languages for monoidal categories. In Coecke, B., editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer Berlin / Heidelberg.

Selinger, P. and Valiron, B. (2006). A lambda calculus for quantum computation with classical control. *MSCS*, 16(3):527–552.

Sen, A. (2006). *The argumentative Indian: Writings on Indian history, culture and identity*. Picador.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656.

Søndergaard, H. and Sestoft, P. (1990). Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517.

Stoddart, B. (2003). Using Forth in an Investigation into Reversible Computation. In *EuroForth*, pages 1–8.

Stoy, J. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.

Strachey, C. (2000). Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49.

Toffoli, T. (1980). Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag.

van Tonder, A. (2004). A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135.

Wadler, P. (1992). The essence of functional programming. In *POPL*, pages 1–14.

Wadler, P. (2003). Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 189–201, New York, NY, USA. ACM.

Wadler, P. (2005). Call-by-value is dual to call-by-name - reloaded. In Giesl, J., editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer.

Walters, R. F. C. (1992). *Categories and Computer Science*. Cambridge University Press.

Weinstein, A. (1996). Groupoids: unifying internal and external symmetry. *Notices of the AMS*, 43(7):744–752.

Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *POPL*, pages 224–235. ACM.

Yokoyama, T., Axelsen, H. B., and Glück, R. (2008). Principles of a reversible programming language. In *Conference on Computing Frontiers*, pages 43–54. ACM.

Yokoyama, T. and Glück, R. (2007). A reversible programming language and its invertible self-interpreter. In *PEPM*, pages 144–153. ACM.

Zeilberger, N. (2010). Polarity and the logic of delimited continuations. *Logic in Computer Science, Symposium on*, 0:219–227.

Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. (2002). Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132.

Zuliani, P. (2001). Logical reversibility. *IBM J. Res. Dev.*, 45:807–818.

Roshan P. James

110 River Drive, Apt 2807, Newport, Jersey City, NJ 07310, USA
Mobile: +425-922-6898
Email: `rpjames@umail.iu.edu, roshan.james@gmail.com`
Homepage: `http://www.cs.indiana.edu/~rpjames`

Academic Qualifications

- *Indiana University, Bloomington*:
  PhD, School of Informatics and Computing, Aug 2007 - May 2013
- *Indiana University, Bloomington*:
  Master of Computer Science, Department of Computer Science, Aug 2005 - May 2007
- *Model Engineering College (MEC), Cochin Univ (CUSAT), Cochin, India*:
  Bachelor of Technology in Computer Engineering, Oct 1998 - Sept, 2002

Selected Papers/Publications

- *Isomorphic Interpreters from Logically Reversible Abstract Machines.*
  Roshan P. James, Amr Sabry *Reversible Computing, RC 2012*
- *Information Effects.*
  Roshan P. James, Amr Sabry *POPL 2012*
- *Embracing the Laws of Physics.*
  Roshan P. James, Amr Sabry *Off the Beaten Track, OBT 2012*
- *Yield: Session Types.*
  Roshan P. James, Amr Sabry *Continuation Workshop, CW 2011*
- *Yield, Mainstream Delimited Continuations.*
  Roshan P. James, Amr Sabry *Theory and Practice of Delimited Continuations, TPDC 2011*
- *Dagger Traced Symmetric Monoidal Categories and Reversible Programming.*
  William J. Bowman, Roshan P. James, Amr Sabry *Reversible Computing, RC 2011*
- *Parallel Generational-Copying Garbage Collection with a Block-Structured Heap.*
  Simon Marlow, Tim Harris, Roshan P. James, Simon Peyton Jones
  International Symposium on Memory Management, ISMM 2008

Full-time Work Experience

- Nov 2013 - current: Software Developer, Jane Street Capital
- Aug 2004 - July 2005: Software Design Engineer, Terminal Services,
  Windows Longhorn/Vista Server,Microsoft Corporation, Hyderabad, India
- Oct 2002 - July 2004: Programmer Analyst, Cognizant Technology Solutions,
  Microsoft Competency Centre, Bangalore, India

Skill Set

- *Development*: OCaml, Scheme, C, C++, C#, Haskell, Ruby
- *Familiar with*: C++/CLI, x86 asm, SQL, Java
- *Operating Systems*: Windows (various), Unix Clones

Summer 2009, 2010: Quant. Intern, Jane Street Capital, NYC and Tokyo

- Worked with Yaron Minsky developing several code development tools, including a distributed computing framework in OCaml based on Hadoop.

Summer 2008: Engineering Intern, Grok project at Google, Kirkland, Washington

- Worked on Steve Yegge's Grok project which attempt to build code analysis and development tools.
- Developed a Parsing Expression Grammar (PEG) based parsing combinator framework for Grok.

Summer 2007: Engineering Intern, Rhino Compiler at Google, Kirkland, Washington

- Worked on Rhino, a JavaScript to JVM bytecode compiler with Bob Jervis and Norris Boyd.
- Major components included a control flow analyzer and implementation of the 'yield' control operator for Rhino

Summer 2006: Research Intern, PL Group, Microsoft Research, Cambridge, UK

- Design and Dev. of an experimental Parallel GC for Haskell (GHC 6.8) with Simon Marlow
- Worked on a call/cc like control operator 'yield' with Simon Peyton Jones

2004-2005: Terminal Services Gateway, Windows Vista Server at Microsoft India

- Design and Development of a gateway/proxy server component of the Windows Longhorn (Vista) Server OS, part of the Terminal Server group.
- Role: Software Design Engineer

2004-2005: Centre for Software Excellence (CSE/BIT) at Microsoft India

- Worked informally on prototypes for code verification using Vulcan and other binary technologies.

2002-2004: Microsoft Competency Center, CTS, Bangalore, India

- Helped form Cognizant's 'Microsoft Competency Centre' at Bangalore
- Role: Work with multiple projects across the company on technical proposals, architectural design, estimations etc.
- Development of internal IP and component infrastructure for use across diverse projects in the company.
- Technologies - .Net and other Microsoft related technologies.

2002-2003: 'Data Definition Language' (DDL)

- Design and Dev. of a unique language for working with binary data formats
- Was designed to be the back engine of a data analysis package for data dumps of Flight Data Recorders for Dimension CyberTech Pvt Ltd, Tvm, India.

2001: Kerala State Network Service: Workflow Mgmt System at Model Engg College

- Design and Dev. of a prototype for file tracking and workflow management service, which highlighted a distributed, decentralized architecture using web-services.
- Developed with guidance from the Microsoft India and Information Kerala Mission (IKM), Kerala State Govt.
- Technologies - C#, ASP.Net, Web Services, ADO.net, SQL Server under Visual Studio .Net Beta 2

Teaching Experience

- P522: Semantics with Amr Sabry.
- P523: Compilers with Kent Dybvig.
- P536: Advanced Operating System with Andrew Lumsdaine.
- P536: Advanced Operating System with Dennis Gannon.
- C212: Introduction to Systems with Amr Sabry.
- A110: Introduction to Computing with Charles Pope.

Other Research Areas and Projects

- Macros and Metaprogramming with Kent Dybvig and Amr Sabry.
- Software Transactional Memory (STM) with Dennis Gannon.
- Compilers and Runtimes: see work on GHC and Rhino.

Undergrad. Technical Merits and Awards

- 2004, 2003 - Most Valuable Professional (MVP) Award by Microsoft Corporation
- 2002 - First - Paper Presentation - IRTT, Erode, Tamil Nadu
- 2002 - First - Software Presentation - IRTT, Erode, Tamil Nadu
- 2002 - First - Software Contest - College of Engineering, Chengannur
- 2002 - Second - Software Contest - NIT , Calicut, India
- 2001 - Second - (National)S/w Devel. - Microsoft Imagine Cup
- 2001 - First - Software Contest - Computer Society of India (CSI), NIT
- 2001 - First - Software Contest - Model Engineering College, Cochin
- 2001 - Second - Software Contest - IEEE, College of Engineering Chengannur