

Information Effects

Roshan P. James
Indiana University
rpjames@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

Computation is a physical process which, like all other physical processes, is fundamentally reversible. From the notion of type isomorphisms, we derive a typed, universal, and reversible computational model in which information is treated as a linear resource that can neither be duplicated nor erased. We use this model as a semantic foundation for computation and show that the “gap” between conventional irreversible computation and logically reversible computation can be captured by a type-and-effect system. Our type-and-effect system is structured as an arrow metalanguage that exposes creation and erasure of information as explicit effect operations. Irreversible computations arise from interactions with an implicit information environment, thus making them a derived notion, much like open systems in Physics. We sketch several applications which can benefit from an explicit treatment of information effects, such as quantitative information-flow security and differential privacy.

Categories and Subject Descriptors D.3.1 [Formal Definitions and Theory]; F.3.2 [Semantics of Programming Languages]; F.3.3 [Studies of Program Constructs]: Type structure

General Terms Languages, Theory

Keywords Arrows, Linear logic, Quantum computing, Reversible logic.

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. Real, quantum-mechanical paper is wildly different from the abstract stuff that the Turing machine uses. The Turing machine is entirely classical, and does not allow for the possibility the paper might have different symbols written on it in different universes, and that those might interfere with one another.” [11, p.252]

The above quote by David Deutsch, originally stated in the context of quantum computing, stems from the observation that *even*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01...\$10.00

the most abstract models of computation embody some laws of Physics. Indeed, 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.

Our main thesis is that by embodying irreversible physical primitives, conventional abstract models of computation have also inadvertently included some *implicit* computational effects, which we call *information effects*. As a consequence of this approach, many applications in which information manipulation is computationally significant are put beyond the reach of our conceptual model of computation. Such applications include quantitative information-flow security [42], differential privacy [14], energy-aware computing [29, 48], VLSI design [30], and biochemical models of computation [9].

In more detail, in Physics, the fundamental laws describe processes in closed systems where every action is reversible. Open systems, which allow irreversible processes, are a derived notion — they can be considered as a subsystem of a closed system which treats the rest of the system as a *global environment*. Pushing the analogy to computation, and as the remainder of the paper formalizes, an irreversible computation can therefore be considered as one that interacts with some global environment via implicit computational effects. Put differently, irreversible computational models like the λ -calculus embody some implicit computational effects which, following the tradition of programming language research, we find useful to expose and study as first-class entities.

Structure and Main Results.

- We develop a pure reversible model of computation that, unlike many other models, does not pre-suppose an existing irreversible model. Our model is obtained from the type isomorphisms and categorical structures that underlie models of linear logic and quantum computing. Technically, the model treats *information* as a linear resource that can neither be erased nor duplicated in a way that is reminiscent of forbidding contraction and weakening in linear logic and the no-cloning and no-deleting theorems of quantum mechanics.¹
- We develop an *arrow metalanguage* that layers effects on top of the pure reversible model above using an arrow abstract type [24]. These arrow effects consist of an explicit *erase* operation that can be used to discard values and an explicit *create* operation which can be used to introduce and duplicate

¹ An embedding of our reversible programming model in Haskell along with several examples can be downloaded from <http://www.cs.indiana.edu/~sabry/papers/Pi.hs>.

values. This construction has the immediate benefit that the information effects are exposed and tracked by the type system.

- We show how to translate a Turing-complete first-order functional language with loops to our reversible model. The translation exposes the implicit erasure and duplication of information in the source language as explicit computational effects. The translation and its associated correctness proof constitute the main technical contribution of the paper.
- We establish connections to, and explore how, the explicit treatment of information effects can benefit applications such as quantitative information-flow security and differential privacy.

2. Logical Reversibility and Information

We review the notion of logical reversibility and its connection to information.

2.1 Reversible Logic

Toffoli’s pioneering work on reversible models of computation [44] established the following fundamental theorem.

THEOREM 2.1 (Toffoli). *For every finite function $\phi : \text{bool}^m \rightarrow \text{bool}^n$ there exists an invertible finite function $\phi^R : \text{bool}^{r+m} \rightarrow \text{bool}^{r+m}$, with $r \leq n$, such that $\phi(x_1, \dots, x_m) = (y_1, \dots, y_n)$ iff*

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

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* : $\text{bool}^2 \rightarrow \text{bool}$ can be compiled to the function *toffoli* : $\text{bool}^3 \rightarrow \text{bool}^3$ which behaves as follows:

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

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

$$\text{toffoli}(v_1, v_2, \text{false}) = \text{if } (v_1 \text{ and } v_2) \text{ then } (v_1, v_2, \text{true}) \text{ else } (v_1, v_2, \text{false})$$

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.

2.2 Thermodynamics of Computation and Information

Toffoli’s work was performed in the context of the work of Landauer [28] and Bennett [7] that established the remarkable result, known as the *Landauer principle*, relating irreversible computations to increase in information uncertainty (entropy).

DEFINITION 2.2 (Entropy of a variable). *Let ‘b’ be a (not necessarily finite) type whose values are labeled b^1, b^2, \dots . 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 $-\sum p_i \log p_i$.*

DEFINITION 2.3 (Output entropy of a function). *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, \dots . 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 2.4. *We say a function is information-preserving if its output entropy is equal to the entropy of its input.*

For example, consider a variable ξ of type $\text{bool} \times \text{bool}$. The information content of this variable depends on the probability distribution of the four possible $\text{bool} \times \text{bool}$ 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 $\text{bool} \times \text{bool}$. The minimum entropy 0 corresponds to a variable that happens to be constant with no uncertainty.

Now consider the $\text{bool} \rightarrow \text{bool}$ 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. As another example, consider the $\text{bool} \rightarrow \text{bool}$ 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. As a final example, consider the $\text{bool} \rightarrow \text{bool} \times \text{bool}$ 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.

2.3 Logical Reversibility

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

DEFINITION 2.5 (Logical reversibility [49]). *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 2.6. *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 $(false, false)$ and $(true, true)$ and in this case it is clear that the function is information-preserving as expected.

Intermezzo. Linear logic [18] 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 $constT'(x) = \text{if } x \text{ then } true \text{ else } true$ 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 [18, pp. 6,17], linear logic embodies a simple and radical change of viewpoint from other logics and this change has a physical flavor.

3. Bijections: Π

Building on the insights of Toffoli, we now turn our attention to defining a logically reversible language with a type structure and a term language. A natural starting point for such a language is the notion of type isomorphisms. In this section, we restrict ourselves to isomorphisms between finite types and show that they naturally lead to a simple programming language which we call Π . In addition to presenting the syntax, type system, and semantics of Π , we establish that Π is universal for reversible combinational circuits.

3.1 Types

The set of finite types b is constructed using sums and products of the primitive type 1. We have the following syntax for types and values:

$$\begin{aligned} \text{value types, } b & ::= 1 \mid b + b \mid b \times b \\ \text{values, } v & ::= () \mid \text{left } v \mid \text{right } v \mid (v, v) \end{aligned}$$

The type 1 has exactly one inhabitant called $()$. Sums allow us to create values that we can distinguish using *left* and *right* constructors and pairs allow the encoding of tuples. Thus we have the type judgements:

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

$$\frac{}{\vdash v : b_1} \quad \frac{}{\vdash v : b_2}{\vdash \text{left } v : b_1 + b_2} \quad \frac{}{\vdash v : b_2}{\vdash \text{right } v : b_1 + b_2}$$

Two types b_1 and b_2 are isomorphic if we can construct a bijective map between their values. The set of sound and complete isomorphisms for finite types is the *congruence closure* of the following primitive isomorphisms [15]:

$$\begin{aligned} b_1 + b_2 & \leftrightarrow b_2 + b_1 \\ b_1 + (b_2 + b_3) & \leftrightarrow (b_1 + b_2) + b_3 \\ 1 \times b & \leftrightarrow b \\ b_1 \times b_2 & \leftrightarrow b_2 \times b_1 \\ b_1 \times (b_2 \times b_3) & \leftrightarrow (b_1 \times b_2) \times b_3 \\ (b_1 + b_2) \times b_3 & \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) \end{aligned}$$

These isomorphisms are already familiar to us from arithmetic or logic (reading 1 as *true*, \times as conjunction, and $+$ as disjunction). Note however that the isomorphisms do not include some familiar logical tautologies, in particular:

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

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.

3.2 Syntax and Semantics

We turn the above isomorphisms into a programming language by associating primitive operators corresponding to the left-to-right and right-to-left reading of each isomorphism. We gather these operators into the table below:

$$\begin{array}{lll} \text{swap}^+ : & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \text{swap}^+ \\ \text{assocl}^+ : & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : \text{assocr}^+ \\ \text{unite} : & 1 \times b \leftrightarrow b & : \text{uniti} \\ \text{swap}^\times : & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : \text{swap}^\times \\ \text{assocl}^\times : & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & : \text{assocr}^\times \\ \text{distrib} : & (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) & : \text{factor} \end{array}$$

Each line of this table is to be read as the definition of one or two operators. For example, the third line declares the two operators $\text{unite} : 1 \times b \leftrightarrow b$ and $\text{uniti} : b \leftrightarrow 1 \times b$. Each of the two cases of commutativity defines one operator that is its own inverse.

Now that we have primitive operators we need some means of composing them. We construct the composition combinators out of the closure conditions for isomorphisms. Thus we have program constructs that witness reflexivity id , symmetry sym , and transitivity \ddagger , and two parallel composition combinators, one for sums and one for pairs.

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

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

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

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

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

swap^+	$(\text{left } v)$	\mapsto	$\text{right } v$
swap^+	$(\text{right } v)$	\mapsto	$\text{left } v$
assocl^+	$(\text{left } v_1)$	\mapsto	$\text{left } (\text{left } v_1)$
assocl^+	$(\text{right } (\text{left } v_2))$	\mapsto	$\text{left } (\text{right } v_2)$
assocl^+	$(\text{right } (\text{right } v_3))$	\mapsto	$\text{right } v_3$
assocr^+	$(\text{left } (\text{left } v_1))$	\mapsto	$\text{left } v_1$
assocr^+	$(\text{left } (\text{right } v_2))$	\mapsto	$\text{right } (\text{left } v_2)$
assocr^+	$(\text{right } v_3)$	\mapsto	$\text{right } (\text{right } v_3)$
unite	$((), v)$	\mapsto	v
uniti	v	\mapsto	$((), v)$
swap^\times	(v_1, v_2)	\mapsto	(v_2, v_1)
assocl^\times	$(v_1, (v_2, v_3))$	\mapsto	$((v_1, v_2), v_3)$
assocr^\times	$((v_1, v_2), v_3)$	\mapsto	$(v_1, (v_2, v_3))$
distrib	$(\text{left } v_1, v_3)$	\mapsto	$\text{left } (v_1, v_3)$
distrib	$(\text{right } v_2, v_3)$	\mapsto	$\text{right } (v_2, v_3)$
factor	$(\text{left } (v_1, v_3))$	\mapsto	$(\text{left } v_1, v_3)$
factor	$(\text{right } (v_2, v_3))$	\mapsto	$(\text{right } v_2, v_3)$

The semantics of composition combinators is:

$$\begin{array}{c}
\frac{}{\text{id } v \mapsto v} \quad \frac{c^\dagger v_1 \mapsto v_2}{(\text{sym } c) v_1 \mapsto v_2} \quad \frac{c_1 v_1 \mapsto v \quad c_2 v \mapsto v_2}{(c_1 \S c_2) v_1 \mapsto v_2} \\
\frac{c_1 v_1 \mapsto v_2}{(c_1 + c_2) (\text{left } v_1) \mapsto \text{left } v_2} \quad \frac{c_2 v_1 \mapsto v_2}{(c_1 + c_2) (\text{right } v_1) \mapsto \text{right } v_2} \\
\frac{c_1 v_1 \mapsto v_3 \quad c_2 v_2 \mapsto v_4}{(c_1 \times c_2) (v_1, v_2) \mapsto (v_3, v_4)}
\end{array}$$

The use of the sym constructor uses the adjoint to reverse the program. We can now verify 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.

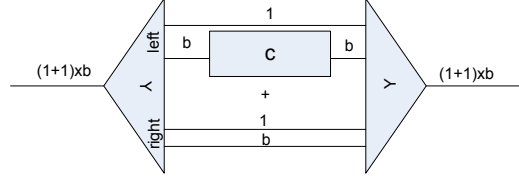
PROPOSITION 3.2 (Logical Reversibility). $c v \mapsto v'$ iff $c^\dagger v' \mapsto v$.

3.3 Expressiveness

There are several universal primitives for conventional (irreversible) hardware circuits (for example, *nand* and *fanout*). In the case of reversible hardware circuits, the canonical universal primitive is the *toffoli* gate (mentioned in Sec. 2.1) which we can express in Π as shown below. Let us start with encoding booleans. The type $1 + 1$ is the type of booleans with *left* $()$ representing *true* and *right* $()$ representing *false*. Boolean negation *not* is simply swap^+ . The Toffoli gate takes three boolean inputs: if the first two inputs are *true* then the third bit is negated. Even though Π lacks conditional expressions, they are expressible using the distributivity laws as we demonstrate. Given any combinator $c : b \leftrightarrow b$ we can construct a combinator called $\text{if}_c : \text{bool} \times b \leftrightarrow \text{bool} \times b$ 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 $\text{distrib } \S ((\text{id} \times c) + \text{id}) \S \text{factor}$.

Let us look at the combinator pictorially as if it were a circuit and values are like particles that flow through this circuit. 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.



We will be seeing many more such wiring diagrams in this paper and it is useful to note some conventions about them. Wires indicate a value that can exist in the program. Each wire, whenever possible, is annotated with its type and sometimes additional information to help clarify its role. When multiple wires run in parallel, it means that those values exist in the system at the same time, indicating pair types. When there is a disjunction, we put a $+$ between the wires. Combinators for distribution distrib and factoring factor are represented as triangles. Other triangles may be used and, in each case, types or labels will be used to clarify their roles. Finally, we don't draw boxes for combinators such as *id*, commutativity, and associativity, but instead just shuffle the wires as appropriate.

The combinator if_{not} has type $\text{bool} \times \text{bool} \leftrightarrow \text{bool} \times \text{bool}$ and negates its second argument if the first argument is *true*. This gate if_{not} is often referred to as the *cnot* gate. If we iterate this construction once more, the resulting combinator if_{cnot} has type $\text{bool} \times (\text{bool} \times \text{bool}) \leftrightarrow \text{bool} \times (\text{bool} \times \text{bool})$. 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 if_{cnot} is the required Toffoli gate.

4. Partial Bijections: Π°

We extend Π with recursive types and a family of looping operators. The resulting language, Π° , is still a language of bijections but, because it can express infinite loops, the bijections may be partial. Despite this extension, the strong version of logical reversibility (Prop. 3.2) still holds for Π° .

4.1 Isorecursive Types and Trace Operators

We 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° , expressive enough to write arbitrary looping programs, including non-terminating ones.

DEFINITION 4.1. (Syntax of Π°) We extend Def. 3.1 as follows:

$$\begin{aligned}
\text{value types, } b & ::= \dots \mid \mu x. b \mid x \\
\text{values, } v & ::= \dots \mid \langle v \rangle \\
\\
\text{combinator types, } t & ::= b \equiv b \\
\text{isomorphisms, iso} & ::= \dots \mid \text{fold} \mid \text{unfold} \\
\text{combinators, } c & ::= \dots \mid \text{trace } c
\end{aligned}$$

The remainder of this section explains the new additions in detail. First, as we illustrate below, it is possible to write infinite loops

in Π° and hence the combinators may correspond to *partial* bijections. We distinguish the type of partial bijections from the previous type of total bijections by using the symbol \rightleftharpoons instead of \leftrightarrow .

Isorecursive Types. The finite types are extended with isorecursive types $\mu x.b$. These 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 all its “unrollings:”

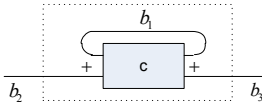
$$\text{fold} : b[\mu x.b/x] \rightleftharpoons \mu x.b \quad \text{unfold}$$

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

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

In other words, to construct a value of type $\mu x.b$, we must first have a value of type $b[\mu x.b/x]$. Depending on the structure of b , this may or not be possible. For example, if the recursive type is $\mu x.x$ then to construct a value of that type, we need to have a value of the same type, ad infinitum. In contrast, if the recursive type is $\mu x.1 + x$, then we can create the initial value *left* () of type $1 + (\mu x.1 + x)$ which leads to the value $\langle \text{left} () \rangle$ of type $\mu x.1 + x$ and then $\langle \text{right} (\text{left} ()) \rangle$, $\langle \text{right} (\text{right} (\text{left} ())) \rangle$ and so on. In fact, the type $\mu x.1 + x$ represents the natural numbers in unary format. The semantics of *fold* and *unfold* is simply $\text{fold } v \mapsto \langle v \rangle$ and $\text{unfold } \langle v \rangle \mapsto v$.

Trace Operators. Traced categories [25] have proved useful for modeling recursion [21]. In the context of Π , the fundamental idea is to add a trace operator with this typing rule:

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


Intuitively, we are given a computation c that accepts a value of type $b_1 + b_2$ and we build a looping version *trace* c that only takes a value of type b_2 . As the diagram illustrates, 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 *trace* c computation. Formally, we can express this semantics as follows:

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

where for each c , loop_c is an internal cyclic version of c .

As before, each combinator has an adjoint and the language is reversible.

PROPOSITION 4.2 (Logical Reversibility). $c v \mapsto v' \text{ iff } c^\dagger v' \mapsto v$.

Proof. The operators *fold* and *unfold* are adjoint to each other. The adjoint of *trace* c is *trace* c^\dagger .

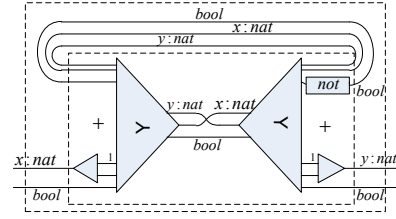
4.2 Expressiveness

We now present several programming examples that illustrate the expressiveness of Π° . These examples are interesting in their own right and are further developed in the accompanying Haskell code which includes implementations of various recursive functions on numbers and lists. Of more immediate relevance however is that

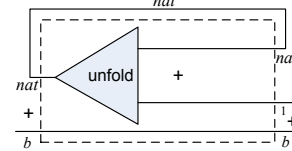
these examples establish idiomatic constructions used to compile conventional source language constructs to Π° as explained in the next sections.

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 examples using diagrams — the full details are included in the accompanying Haskell code.

Bounded Iteration. The simplest class of examples takes a number n and iterates a particular combinator n times. For example, it is possible to write a function *even?* : $\text{nat} \times \text{bool} \rightleftharpoons \text{nat} \times \text{bool}$ which, given inputs (n, b) , reveals whether n is even or odd by iterating *not* n -times starting with b . The iteration is realized using *trace* as shown in the diagram below (where we have omitted the boxes for *fold* and *unfold*):



Partiality. The combinator *just* : $b \rightleftharpoons 1 + b$ below injects a value into a larger type. This combinator is significant because it shows that Π° admits non-terminating computations as the adjoint of *just* diverges on *left* ():



Using *just*, we can conveniently write *add*₁ and *sub*₁ as *add*₁ = *just* ; *fold* and *sub*₁ = *sym* *add*₁. (The definition implies that *sub*₁ 0 diverges.) 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:

$$\begin{aligned} \text{introF}, \text{introT} : 1 &\rightleftharpoons \text{bool} \\ \text{introF} &= \text{just} \\ \text{introT} &= \text{just} \ ; \ \text{not} \end{aligned}$$

Given these functions, we can inject a value into a left or right summand. For example,

$$\begin{aligned} \text{injectR} : a &\rightleftharpoons a + a \\ \text{injectR} &= \text{uniti} \ ; \ (\text{introF} \times \text{id}) \ ; \ \text{distrib} \ ; \ (\text{unite} + \text{unite}) \end{aligned}$$

We can introduce 0 as follows:

$$\begin{aligned} \text{introZ} : 1 &\rightleftharpoons \text{nat} \\ \text{introZ} &= \text{trace} (\text{swap}^+ \ ; \ \text{fold} \ ; \ \text{injectR}) \end{aligned}$$

Similarly, we can also introduce an empty list of any type. More precisely, let the encoding of lists be $[b] \equiv \mu x.1 + b \times x$. Given a type b and a combinator *introConst* _{b} to introduce a constant of type b , we can write *introNil* : $1 \equiv [b]$ which introduces an empty list of type $[b]$. Adding an element to a list can be achieved using a construction that is similar to *add*₁; accessing the head and tail of a list can be realized using constructions that are similar to *sub*₁.

5. Source Language

Having designed Π° as a language that embodies the physical idea of reversibility, we now wish to demonstrate how irreversible programming language constructs correspond to open systems which implicitly communicate with a global heap and garbage dump. To make this idea concrete, we need a canonical irreversible language.

We use for that purpose a simply-typed, first-order functional language with sums and pairs and **for** loops. The language is fairly conventional and is presented without much discussion.

We present this language as two fragments: the first of these, LET, is strongly normalizing, and the second, LET^o, includes natural numbers and **for** loops for iteration.

The syntax of LET is given below:

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

The most interesting aspect of LET is that expressions may freely erase and duplicate data in irreversible ways.

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

$$\begin{array}{lcl}
\text{Base types, } b & = & \dots \mid \text{nat} \\
\text{Values, } v & = & \dots \mid n \\
\text{Expressions, } e & = & \dots \mid n \mid \text{add}_1 e \mid \text{sub}_1 e \mid \text{iszero? } e \\
& & \mid \text{for } x = e_1 \text{ if } e_2 \text{ do } e_3
\end{array}$$

The most interesting aspect of the extended language LET^o is that it admits partial functions. Its type system is entirely conventional except for the rule below:

$$\frac{\Gamma \vdash e_1 : b \quad \Gamma, x : b \vdash e_2 : \text{bool} \quad \Gamma, x : b \vdash e_3 : b}{\Gamma \vdash \text{for } x = e_1 \text{ if } e_2 \text{ do } e_3 : b}$$

Here is an example of iterative addition of two numbers n and m in this syntax:

$$\text{snd } (\text{for } x = (n, m) \text{ if not } (\text{iszero? } (\text{fst } x)) \\
\text{do } (\text{sub}_1 (\text{fst } x), \text{add}_1 (\text{snd } x)))$$

The *not* operator over the type $\text{bool} = 1 + 1$ can be macro encoded as $\text{not } x = \text{case } x \text{ [y.right ()] [y.left ()]}$.

For the semantics, we say $\text{eval}_{\text{let}}(e) = v$ if $\langle e, \epsilon \rangle \mapsto_{\text{let}}^* v$ according to a conventional big-step relation mapping closed expressions and environments to values, of which we show a few representative cases below:

$$\begin{array}{c}
\frac{\rho(x) = v \quad \langle e_1, \rho \rangle \mapsto_{\text{let}} v_1 \quad \langle e_2, \rho; x = v_1 \rangle \mapsto_{\text{let}} v_2}{\langle x, \rho \rangle \mapsto_{\text{let}} v} \quad \langle \text{let } x = e_1 \text{ in } e_2, \rho \rangle \mapsto_{\text{let}} v_2 \\
\frac{\langle e, \rho \rangle \mapsto_{\text{let}} \text{left } v_1 \quad \langle e_1, \rho; x = v_1 \rangle \mapsto_{\text{let}} v}{\langle \text{case } e \text{ x.} e_1 \text{ x.} e_2, \rho \rangle \mapsto_{\text{let}} v} \\
\frac{\langle e_1, \rho \rangle \mapsto_{\text{let}} v_1 \quad \langle e_2, \rho \rangle \mapsto_{\text{let}} v_2}{\langle (e_1, e_2), \rho \rangle \mapsto_{\text{let}} (v_1, v_2)} \quad \frac{\langle e_1, \rho \rangle \mapsto_{\text{let}} v \quad \langle e_2, \rho; x = v \rangle \mapsto_{\text{let}} \text{false}}{\langle \text{for } x = e_1 \text{ if } e_2 \text{ do } e_3, \rho \rangle \mapsto_{\text{let}} v} \\
\frac{\langle e_1, \rho \rangle \mapsto_{\text{let}} v \quad \langle e_2, \rho; x = v \rangle \mapsto_{\text{let}} \text{true}}{\langle \text{for } x = e_3 \text{ if } e_2 \text{ do } e_3, \rho; x = v \rangle \mapsto_{\text{let}} v'} \\
\langle \text{for } x = e_1 \text{ if } e_2 \text{ do } e_3, \rho \rangle \mapsto_{\text{let}} v'
\end{array}$$

6. Metalanguages and Translations: An Overview

The technical goal of the next four sections (Secs. 7 to 10) is to translate the source language LET^o with irreversible primitives to the target information-preserving language Π^o .

Arrow Metalanguage. The first important point is that the translation is defined via a metalanguage for information effects. 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 [36] in the translation of conventional computational effects (e.g., control operators [22]). Our metalanguage is defined, not by extending the λ -calculus with monadic

combinators, but by extending Π^o with arrow combinators. Starting with Π^o instead of the λ -calculus is expected since Π^o plays the role of the pure effect-free language in our setting, just like the λ -calculus plays the role of the pure effect-free language in the conventional setting. The choice of using the arrow combinators rather than the monadic combinators is because the notion of information effects does not appear to be expressible as a monad. For the purposes of presentation, the metalanguage is defined in two stages: ML_{Π} in Sec. 7 which is used to compile the strongly-normalizing subset of the source language using two effect combinators **create** and **erase**, and ML_{Π^o} in Sec. 10.1 which is used to compile the full source language using an additional effect combinator **traceA**. Thus, Secs. 7, 8 and 9 focus on the translation from LET to Π and Sec. 10 handles the translation from LET^o to Π^o .

Translation from Source to Metalanguage. 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. 5) 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. Technically, we define two translations $\mathcal{T}_1 : \text{LET} \Rightarrow \text{ML}_{\Pi}$ and $\mathcal{T}_1^o : \text{LET}^o \Rightarrow \text{ML}_{\Pi^o}$. The first maps the strongly-normalizing subset of the source to ML_{Π} (Sec. 8) and the second is an extension that handles the full source and targets ML_{Π^o} (Sec. 10.2).

Translation from Metalanguage to Target. This translation essentially needs to compile the effect combinators to the target language. The basic scheme is based on Toffoli's idea described in Sec. 2.1: an irreversible function of type $a \rightarrow b$ 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. Once the primitive effect combinators have been translated, the arrow combinators then 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 $\mathcal{T}_2 :: \text{ML}_{\Pi} \Rightarrow \Pi$ (Sec. 9) selects particular values for the heap and garbage to embed the irreversible effects into bijections.² The second translation $\mathcal{T}_2^o :: \text{ML}_{\Pi^o} \Rightarrow \Pi^o$ (Sec. 10.3) 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 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. 4.2, Π^o can express the creation and erasure of constants and we leverage this to completely eliminate the heap in favor of using **traceA**.³

7. Arrow Metalanguage ML_{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. In particular,

²This idea has been the basis of translations similar to ours [5]. The literature also includes translations for other languages [2, 12, 19, 23, 26, 41, 49] that share some of the intuition of the translation we present but differ significantly in the technical details.

³If 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.

we add two operators `create` and `erase` which correspond to the creation and erasure of information that is implicit in the semantics of LET. These operators are not isomorphisms and hence cannot have \leftrightarrow types. They can only have arrow types.

DEFINITION 7.1. (Syntax of ML_{Π}) The sets of value types, values, and isomorphisms are identical to the corresponding sets in Π (see Def. 3.1). The extended combinator types and arrow computations are defined as follows:

$$\begin{aligned} \text{types, } t &::= b \leftrightarrow b \mid b \rightsquigarrow b \\ \text{arrow comp., } a &::= \text{iso} \mid a + a \mid a \times a \mid a \S a \\ &\quad \mid \text{arr } a \mid a \ggg a \mid \text{first } a \mid \text{left } a \\ &\quad \mid \text{create}_b \mid \text{erase} \end{aligned}$$

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 `createb` 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:

$$\frac{a : b_1 \leftrightarrow b_2}{\text{arr } a : b_1 \rightsquigarrow b_2} \quad \frac{a_1 : b_1 \rightsquigarrow b_2 \quad a_2 : b_2 \rightsquigarrow b_3}{a_1 \ggg a_2 : b_1 \rightsquigarrow b_3} \quad \frac{a : b_1 \rightsquigarrow b_2}{\text{first } a : b_1 \times b_3 \rightsquigarrow b_2 \times b_3} \quad \frac{a : b_1 \rightsquigarrow b_2}{\text{left } a : b_1 + b_3 \rightsquigarrow b_2 + b_3}$$

$$\text{create}_b : 1 \rightsquigarrow b \quad \text{erase} : b \rightsquigarrow 1$$

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:

$$\frac{a v_1 \mapsto v_2}{(\text{arr } a) v_1 \mapsto_{ML} v_2} \quad \frac{a_1 v_1 \mapsto_{ML} v_2 \quad a_2 v_2 \mapsto_{ML} v_3}{(a_1 \ggg a_2) v_1 \mapsto_{ML} v_3} \quad \frac{a v_1 \mapsto_{ML} v_2}{(\text{left } a) (\text{left } v_1) \mapsto_{ML} \text{left } v_2} \quad \frac{a v_1 \mapsto_{ML} v_2}{(\text{left } a) (\text{right } v) \mapsto_{ML} \text{right } v}$$

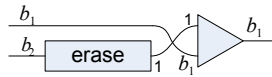
$$\frac{}{(\text{first } a) (v_1, v_3) \mapsto_{ML} (v_2, v_3)} \quad \frac{}{\text{erase } v \mapsto_{ML} ()} \quad \frac{}{\text{create}_b() \mapsto_{ML} \phi(b)}$$

The operator `erase` at type b takes any value of type b and returns $()$ which contains no information. For any type b , `createb` returns a fixed (but arbitrary) value of type b which we call $\phi(b)$ and which is defined as:

$$\phi(1) = () \quad \phi(b_1 \times b_2) = (\phi(b_1), \phi(b_2)) \quad \phi(b_1 + b_2) = \text{left}(\phi(b_1))$$

The two operators `createb` 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).

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 `(arr unite) : 1 \times b \rightsquigarrow b` to absorb the $()$. Thus we have:



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

$$\text{fstA} = \text{second } \text{erase} \ggg \text{arr} (\text{swap}^\times \S \text{unite})$$

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

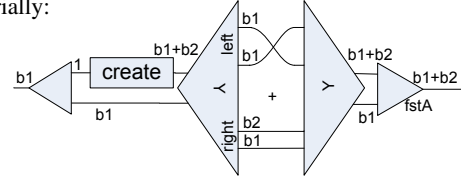
Injecting a value in a larger type (leftA). The combinator `leftA` : $b_1 \rightsquigarrow b_1 + b_2$ takes a value and injects it in a larger sum type. Its definition is involved so we begin by defining the following combinator in Π :

$$\begin{aligned} \text{leftSwap} &: (b_1 + b_2) \times b_1 \leftrightarrow (b_1 + b_2) \times b_1 \\ \text{leftSwap} &= (\text{distrib} \S (\text{swap}^\times + \text{id}) \S \text{factor}) \end{aligned}$$

Given values v_1 and v'_1 both of type b_1 , the application of `leftSwap` to $(\text{left } v_1, v'_1)$ produces $(\text{left } v'_1, v_1)$, 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 $(\text{left } v'_1, v_1)$, and complete the definition by using `fstA` to erase the default value v_1 . Thus `leftA` is:

$$\begin{aligned} &(\text{arr } \text{unite}) \ggg (\text{first } \text{create}_{b_1+b_2}) \\ &\ggg (\text{arr } \text{leftSwap}) \ggg \text{fstA} \end{aligned}$$

or pictorially:



The symmetric combinator `rightA` can be defined similarly.

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:

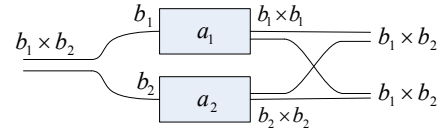
$$((\text{arr } \text{unite}) \oplus (\text{arr } \text{unite})) \ggg (\text{arr } \text{factor}) \ggg \text{sndA}$$

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

LEMMA 7.2 (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 $a = \text{arr } \text{unite}$
- 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:



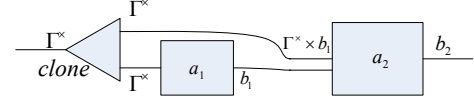
Thus we can write $a = (a_1 \otimes a_2) \ggg \text{shuffle}$ where `shuffle` is the operator which rearranges the pairs as pictured.

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



This combinator clones b_1 and then applies `leftA` to one of the copies. Let us call this combinator $a'_1 : b_1 \rightsquigarrow b_1 \times (b_1 + b_2)$. We

can do the same with b_2 , except that we apply `rightA`, resulting in combinator $a'_2 : b_2 \rightsquigarrow b_2 \times (b_1 + b_2)$. The required combinator a can be constructed by applying these in parallel and factoring out the results, i.e., $a = (a'_1 \oplus a'_2) \ggg (\text{arr factor})$



8. Translation from LET to ML_Π

The translation \mathcal{T}_1 maps a closed term of type b in LET to an ML_Π 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.

8.1 Environments

A LET type environment Γ is translated to an ML_Π type as follows:

$$\begin{aligned} [\epsilon]^\times &= 1 \\ [\Gamma, x : b]^\times &= [\Gamma]^\times \times b \end{aligned}$$

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

$$\begin{aligned} [\epsilon]^\times &= () \\ [\rho, x = v]^\times &= ([\rho]^\times, v) \end{aligned}$$

LEMMA 8.1 (Lookup). *If $\Gamma \vdash x : b$ and Γ^\times is the encoding of Γ , then there exists a combinator $a_{\text{lookup}(x)} : \Gamma^\times \rightsquigarrow b$ that looks up x in Γ^\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 $a = \text{sndA}$
- 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 $a = (a' \otimes \text{id}) \ggg \text{fstA}$.

8.2 The Translation \mathcal{T}_1

We translate a LET judgment of the form $\Gamma \vdash e : b$ to an ML_Π combinator $a : \Gamma^\times \rightsquigarrow b$ in such a way that the execution of the resulting ML_Π 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 Γ^\times denoting the value of the environment.

LEMMA 8.2 (\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 Γ^\times in ML_Π such that:*

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

We simultaneously present the translation and prove its correctness:

- Case $()$:

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

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

- Case x :

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

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

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

To construct the required a we first clone Γ^\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:

$$a = \text{clone}_{\Gamma^\times} \ggg (\text{second } a_1) \ggg a_2$$

- Case (e_1, e_2) :

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

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

$$a = \text{clone}_{\Gamma^\times} \ggg (a_1 \otimes a_2)$$

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

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

We have $a = a_1 \ggg \text{fstA}$. And similarly for *snd* e we have $a = a_1 \ggg \text{sndA}$.

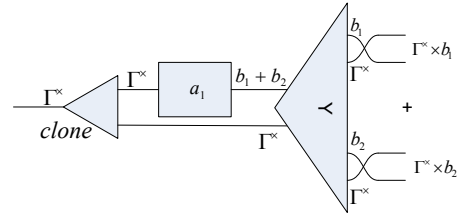
- Cases *left* e , *right* e :

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

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

- Case *case* e_1 $x.e_2$ $x.e_3$:

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



Here we have cloned Γ^\times and constructed $b_1 + b_2$ using one copy of Γ^\times and a_1 . We then distributed Γ^\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:

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

9. Translation from ML_Π to Π

The translation \mathcal{T}_2 maps an ML_Π 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 9.1 (\mathcal{T}_2 and its correctness). *For any ML_{Π} 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 :

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

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

- $a_1 \ggg a_2$:

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

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

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

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

- **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 $c = \mathit{assocl}^\times \mathbin{\&} (c_1 \times id) \mathbin{\&} \mathit{assocr}^\times$

- **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:

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

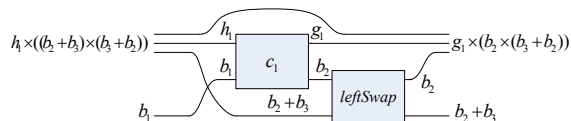
$$g'' = (h_1 \times ((b_2 + b_3) \times b_3))$$

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. 7. We can draw a wiring diagram for c' as follows:

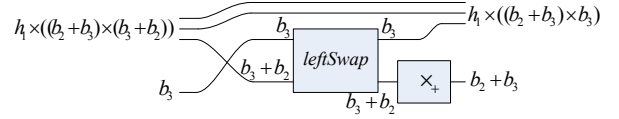


The combinator c' uses c_1 to obtain a value of type b_2 and then uses *leftSwap* to construct a value of type $b_2 + b_3$. We can now define c'' that works on the *right* branch to be:

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

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

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:

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

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

- **create**:

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

We choose $h = b$ and $g = 1$ and we have $c = \mathit{swap}^\times$. 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**:

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

We choose $h = 1$ and $g = b$ and we have $c = \mathit{swap}^\times$. 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.

10. Translation from LET° to Π°

The required translation \mathcal{T}° is factored into $\mathcal{T}_1^\circ : LET^\circ \implies ML_{\Pi^\circ}$ and $\mathcal{T}_2^\circ : ML_{\Pi^\circ} \implies \Pi^\circ$ where the intermediate language ML_{Π° extends ML_{Π} .

10.1 Arrow Metalanguage : ML_{Π°

The arrow metalanguage, ML_{Π° , extends ML_{Π} with natural numbers and loops.

DEFINITION 10.1 (Syntax of ML_{Π°).

<i>base types, b</i>	=	$1 \mid b \times b \mid b + b \mid \mathit{nat}$
<i>values, v</i>	=	$() \mid (v, v) \mid \mathit{left} v \mid \mathit{right} v \mid n$
<i>types, t</i>	::=	$b \equiv b \mid b \rightarrow b$
<i>arrow comp., a</i>	::=	$\mathit{iso} \mid a + a \mid a \times a \mid a \mathbin{\&} a \mid \mathit{trace} a$
		$\mid \mathbf{arr} a \mid a \ggg a \mid \mathbf{first} a \mid \mathbf{left} a$
		$\mid \mathbf{trace}_b A \mid \mathbf{create}_b \mid \mathbf{erase}$

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* $()$. The arrow type \rightarrow is analogous to \rightsquigarrow of ML_{Π} : we use a different symbol to emphasize that the underlying bijections are partial. The set of underlying isomorphisms extends the ones for finite types with *unfold* : $\mathit{nat} \equiv 1 + \mathit{nat}$: *fold*. We define $\phi(\mathit{nat}) = 0$ and hence $\mathbf{create}_{\mathit{nat}} = 0$. Finally, the language includes one additional arrow combinator **trace**_A with the typing rule:

$$\frac{a : b_1 + b_2 \rightarrow b_1 + b_3}{\text{traceA } a : b_2 \rightarrow b_3}$$

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 semantics of *traceA* is similar to that of *trace* but with an important technical difference. The body of the *traceA*-loop may produce garbage values which *traceA* collects in a list as the iteration progresses. (See the last case of the translation \mathcal{T}_2° below.) As before we can clone any particular value.

LEMMA 10.2 (Cloning). *For any type b , we can construct an operator clone of type $b \rightarrow b \times b$ such that: clone $v \mapsto_{ML} (v, v)$*

Proof. This is an extension of Lemma 7.2. The only new datatype is *nat*. We *create* a 0 and use the loop of Sec. 4.2 to iteratively *add*₁.

10.2 Translation \mathcal{T}_1° from LET^o to ML_{Π^o}

The translation \mathcal{T}_1° extends the translation \mathcal{T}_1 from LET to ML_Π.

LEMMA 10.3 (\mathcal{T}_1° and its correctness). *For any well typed LET^o expression $\Gamma \vdash e : b$, $\mathcal{T}_1^\circ[\Gamma \vdash e : b]$ gives us a combinator 'a' and a type Γ^\times in ML_{Π^o} such that:*

1. $a : \Gamma^\times \rightarrow b$
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. 4.2.

$$\frac{\Gamma \vdash e : \text{nat} \rightarrow a : \Gamma^\times \rightarrow \text{nat}}{\Gamma \vdash \text{add}_1 e : \text{nat} \rightarrow a \ ; \ (\text{arr } \text{add}_1) : \Gamma^\times \rightarrow \text{nat}}$$

- Case *iszero?* *e*:

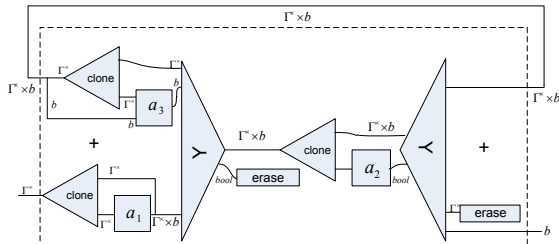
$$\frac{\Gamma \vdash e : \text{nat} \rightarrow a_1 : \Gamma^\times \rightarrow \text{nat}}{\Gamma \vdash \text{iszero? } e : \text{nat} \rightarrow a : \Gamma^\times \rightarrow \text{bool}}$$

where $a = a_1 \ ; \ \text{arr } \text{unfold} \ ; \ \text{first } \text{erase} \ ; \ \text{arr } \text{unite}$

- Case *n*: The required combinator of type $\Gamma \rightarrow \text{nat}$ is given by *erase* ; *create*_{nat} ; *add*₁ⁿ (where *add*₁ⁿ is *n* iterations of *add*₁).
- Case **for** $x = e_1$ **if** e_2 **do** e_3 :

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : b \rightarrow a_1 : \Gamma^\times \rightarrow b \\ \Gamma, x : b \vdash e_2 : \text{bool} \rightarrow a_2 : \Gamma^\times \times b \rightarrow \text{bool} \\ \Gamma, x : b \vdash e_3 : b \rightarrow a_3 : \Gamma^\times \times b \rightarrow b \end{array}}{\Gamma \vdash \text{for } x = e_1 \text{ if } e_2 \text{ do } e_3 : b \rightarrow a : \Gamma^\times \rightarrow b}$$

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.

10.3 Translation \mathcal{T}_2° from ML_{Π^o} to Π^o

Translation \mathcal{T}_2° is similar to \mathcal{T}_2 . As discussed in the overview (Sec. 6) 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 10.4. *For any type b of ML_{Π^o} we can construct $\text{createConst}_b : 1 \Rightarrow b$ such that $\text{createConst}_b () \mapsto_{ML} \phi_b$.*

LEMMA 10.5 (\mathcal{T}_2° and its correctness). *For any ML_{Π^o} combinator $a : b_1 \rightarrow b_2$, $\mathcal{T}_2^\circ[a : b_1 \rightarrow b_2]$ gives us c and g in Π^o such that:*

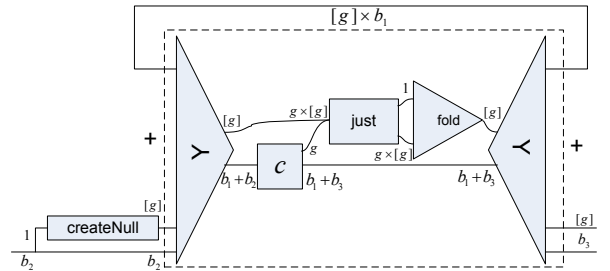
- $c : b_1 \Rightarrow g \times b_2$
- $\forall (v_1 : b_1), \exists (v_g : g), (v_2 : b_2)$. if $a v \mapsto_{ML}^* v_2$ then $c v_1 \mapsto^* (v_g, v_2)$.

The interesting cases to consider are:

- Case *arr* *c*: The required combinator is $c \ ; \ \text{unit}_i : b_1 \Rightarrow 1 \times b_2$ where the garbage is 1.
- Case *create*_{*b*}: The required combinator is $\text{createConst}_b \ ; \ \text{unit}_i : 1 \Rightarrow 1 \times b$ with $g = 1$.
- Case *erase*: The required combinator is $\text{unit}_i \ ; \ \text{swap}^\times : 1 \Rightarrow b \times 1$ with $g = b$.
- Case *traceA* *a*:

$$\frac{a : b_1 + b_2 \rightarrow b_1 + b_3 \rightarrow c : b_1 + b_2 \Rightarrow g \times (b_1 + b_3)}{\text{traceA } a : b_2 \rightarrow b_3 \rightarrow c_1 : b_2 \Rightarrow g' \times b_3}$$

As shown in Sec. 4.2, 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 .



11. Applications

The technical contribution of the paper was devoted to designing a semantic foundation of computation which treats information as a computational resource. We now briefly explore the application of such a framework to security and privacy.

11.1 Quantitative Information Flow

Research in the domain of quantitative information-flow security is aimed at tracking the amount of information leakage through a computation [10, 32, 33]. 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 takes a 2-bit user input and has access to a 2-bit secret password. The type $bool \times bool$ has 4 inhabitants. 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 $1/4 \log 4 + 3/4(\log 4 - \log 3) = 0.8$. One call to 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 by some logically irreversible erasure of 1.2 bits. And indeed, the minimum any Π° implementation of $check$ must $erase$ is 2 bits which would manifest itself by a use of $erase$ at type $bool \times bool$. The fact that ML_{Π° types indicate a lower bound on a program’s intrinsic secrecy deserves further investigation. 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° to track the probability of disjunctive branches, such as in the probabilistic λ -calculus [39].

11.2 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. Toffoli and Fredkin [16] captured this additional restriction on physical processes using what they called “Conservative Logic” in which values can only be shuffled around by computation. This guarantees that processes maintain whatever cost is associated with the values they operate on. 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. Their key definition is the following.

DEFINITION 11.1 (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 [40] 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. Their key definition is the following.

DEFINITION 11.2 (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.

12. Conclusions and Future Work

Starting with the notion that closed physical systems are the “purest,” we have developed a pure language in which computations preserve information. 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. One of our main contributions therefore is in bridging 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.

The main thesis of the paper is that 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. Our paper provides the conceptual foundation for such investigations.

Reversible Computing. Our pure language is logically reversible and hence shares some common features with previously developed reversible languages [6, 34, 37, 47] although none are based on the simple notion of isomorphisms between types [8]. The practice of programming languages is replete with *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. 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. Similarly, our wiring diagrams could be directly realized as hardware circuits and pave the way for speculative execution and backtracking infrastructure in CPUs that are inherently reversible and use minimal bookkeeping resources [20].

Quantum Computing. Many programming models of quantum computing start with the λ -calculus as the underlying classical language and add quantum features on top of it [5, 13, 43, 45]. 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 their implicit information effects. A simpler strategy that loses no generality is to build the quantum features on top of a reversible classical language.

Optimality and Equivalence Preservation. For our purpose of establishing a semantic connection, we have made no attempt to optimize the types h and g generated by the translation to the reversible target language. For applications concerned with the implementation of Π° circuits, optimizations like in Toffoli’s original paper will need to be developed.

Also, 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. 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 [4, 35].

Higher-Order Functions. As the development of lambda calculi with linear [27, 46] and bunched types [38] shows, controlling the creation, duplication, and sharing of resources is largely orthogonal to higher-order functions. Furthermore, existing work suggests that adding higher-order functions to a language like Π° can be done in a systematic way as shown for example by the ‘Int construction’ of Joyal, Street and Verity in the context of traced monoidal categories [25], or by Abramsky [1, 3] and Mackie [31] in the context of the geometry of interaction and Ghica [17] in the context of the geometry of synthesis. These constructions need to be explored in the context of Π° .

Acknowledgments

We thank Amal Ahmed, Esfandiar Haghverdi, and Erik Wennstrom for helpful discussions. We also thank the anonymous reviewers for their helpful comments and suggestions. This project was partially funded by Indiana University's Office of the Vice President for Research and the Office of the Vice Provost for Research through its Faculty Research Support Program. We also acknowledge support from Indiana University's Institute for Advanced Study.

References

- [1] S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.
- [2] S. Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347:441–464, December 2005.
- [3] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Inf. Comput.*, 111:53–119, May 1994.
- [4] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, pages 157–168. ACM, 2008.
- [5] T. Altenkirch and J. Grattage. A functional quantum programming language. In P. Panangaden, editor, *LICS*, pages 249–258. IEEE Computer Society Press, June 2005.
- [6] H. G. Baker. NREVERSAL of fortune - the thermodynamics of garbage collection. In *Proceedings of the International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.
- [7] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532, November 1973.
- [8] W. J. Bowman, R. P. James, and A. Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Workshop on Reversible Computation*, 2011.
- [9] L. Cardelli and G. Zavattaro. On the computational power of biochemistry. In *Third International Conference on Algebraic Biology*, 2008.
- [10] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15:321–371, August 2007.
- [11] D. Deutsch. *The Fabric of Reality*. Penguin, 1997.
- [12] A. Di Pierro, C. Hankin, and H. Wiklicky. Reversible combinatory logic. *MSCS*, 16:621–637, August 2006.
- [13] R. Duncan. *Types for quantum computation*. PhD thesis, Oxford University, 2006.
- [14] C. Dwork. Differential privacy. In *ICALP (2)'06*, pages 1–12, 2006.
- [15] M. Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.
- [16] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.
- [17] D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375. ACM, 2007.
- [18] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
- [19] R. Glück and M. Kawabe. Revisiting an automatic program inverter for Lisp. *SIGPLAN Not.*, 40:8–17, May 2005.
- [20] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, pages 58–69, New York, NY, USA, 1998. ACM.
- [21] M. Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*, pages 196–213. Springer-Verlag, 1997.
- [22] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *POPL*, pages 458–471. ACM, 1994.
- [23] L. Huelsbergen. A logically reversible evaluator for the call-by-name lambda calculus. *InterJournal Complex Systems*, 46, 1996.
- [24] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.
- [25] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Philos. Soc.*, 119(3):447–468, 1996.
- [26] W. E. Kluge. A reversible SE(M)CD machine. In *International Workshop on Implementation of Functional Languages*, pages 95–113. Springer-Verlag, 2000.
- [27] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59:157–180, July 1988. ISSN 0304-3975.
- [28] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, July 1961.
- [29] X. Ma, J. Huang, and F. Lombardi. A model for computing and energy dissipation of molecular QCA devices and circuits. *J. Emerg. Technol. Comput. Syst.*, 3(4):1–30, 2008.
- [30] E. Macii and M. Poncino. Exact computation of the entropy of a logic circuit. In *Proceedings of the 6th Great Lakes Symposium on VLSI*, Washington, DC, USA, 1996. IEEE Computer Society.
- [31] I. Mackie. Reversible higher-order computations. In *Workshop on Reversible Computation*, 2011.
- [32] P. Malacaria. Assessing security threats of looping constructs. In *POPL*, pages 225–235. ACM, 2007.
- [33] J. L. Massey. Guessing and entropy. In *Proceedings of the IEEE International Symposium on Information Theory*, page 204, 1994.
- [34] A. B. Matos. Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290:2063–2074, January 2003.
- [35] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *POPL*, pages 271–283, New York, NY, USA, 1996. ACM.
- [36] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.
- [37] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *International Conference on Mathematics of Program Construction*, pages 289–313, 2004.
- [38] P. O’hearn. On bunched typing. *J. Funct. Program.*, 13:747–796, July 2003.
- [39] A. D. Pierro, C. Hankin, and H. Wiklicky. Probabilistic lambda-calculus and quantitative program analysis. *J. Log. Comput.*, 15(2), 2005.
- [40] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ICFP*, pages 157–168. ACM, 2010.
- [41] B. J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1997.
- [42] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [43] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *MSCS*, 16(3):527–552, 2006.
- [44] T. Toffoli. Reversible computing. In *Proceedings of the Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.
- [45] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004.
- [46] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.
- [47] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Conference on Computing Frontiers*, pages 43–54. ACM, 2008.
- [48] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.
- [49] P. Zuliani. Logical reversibility. *IBM J. Res. Dev.*, 45:807–818, November 2001.