Flow Typing for Lightweight Linearity

SKY WILSHAW, University of Nottingham, United Kingdom
GRAHAM HUTTON, University of Nottingham, United Kingdom

Linear types are useful for writing safe resource-sensitive programs, but a strict linearity discipline is too burdensome to uphold in practice. To make linearity practical, languages often add extra typing features such as borrowing, but such features complicate the semantics and can be challenging to use. In this paper, we introduce *flow typing* as an alternative approach. Flow typing allows us to write linear code in a natural and familiar style, but has a simple semantics via a translation into a linear lambda calculus. We introduce and implement our ideas using a Haskell-like language that is compiled down to Linear Haskell.

1 Introduction

Linear types can be used to write resource-sensitive programs [33]. A type is called *linear* if its values cannot be freely duplicated or discarded, corresponding to the fact that these operations may not always be applicable to real-world resources. Linearity is beneficial because it allows us to write safer and more efficient programs. For example, it can be used to control access to shared resources and to permit in-place updates. In addition, linearity allows for more granular tracking of permissions and state, enabling concepts such as typestate [30] and session types [15].

One key feature of linear types is that any operation that uses a resource must consume it or return it. Unfortunately, this is true even for non-destructive operations. For example, the following Haskell-like pseudocode computes the length of a list of linear values, but must simultaneously rebuild and return the list so that the underlying resources are not discarded:

```
length :: [a] -> (Int, [a])
length []     = (0, [])
length (x:xs) = let (n, ys) = length xs in (n+1, x:ys)
```

This code is much less clear than an equivalent non-linear program that simply returns the length, and manually rebuilding data structures in this way is tedious and error-prone. How can we retain the benefits of linear types without incurring this additional complexity?

One natural approach to try to simplify the above code is to wrap the computation in a state monad, where the list forms the state. However, the usual state monad operations `get` and `put` are not linear, so they cannot be written in a linear type theory. For example, `get` duplicates the state, returning one copy and keeping the other. There are workarounds for this problem, such as using custom combinators that manipulate state transformers in a linear manner. However, the use of such combinators requires writing programs in a rather unnatural style.

A second approach is to add a nonlinear type of references to the type system. This would allow us to compute the length of a list by reference, without actually consuming the underlying resources. In Rust, for example, values can be temporarily *borrowed* to obtain a reference, and after the borrow has ended, the original value can be used again [21]. Other proposals for adding references into linear languages include fractional uniqueness types [20], implemented in the Granule language, which allow for a functional presentation of borrowing. These ideas use sophisticated type system features to ensure that linearity is not violated, but this adds significant complexity. Indeed, the study of Zhu et al. [36] found that "[Rust's] complex safety rules pose unique difficulties" for programmers, which reflects the well-known experience of 'fighting the borrow checker'.

Authors' Contact Information: Sky Wilshaw, School of Computer Science, University of Nottingham, United Kingdom, sky.wilshaw@nottingham.ac.uk; Graham Hutton, School of Computer Science, University of Nottingham, United Kingdom, graham.hutton@nottingham.ac.uk.

In this article, we present an alternative approach, introducing a linear type theory that allows functions to be defined in a natural and familiar style while retaining the benefits of linear types. To achieve this, we treat the typing context that records the type of each variable as a *state parameter*, threading it through the type checking process. We call this paradigm *flow typing*. The key point of flow typing is that it supports a way to use variables 'by reference'. In a conventional linear programming language, referring to a variable removes it from the context, so other parts of the program cannot use it. With flow typing, however, we can *temporarily* remove a variable from the context, use it, and then return it to the context. We use the phrase 'by reference' to describe such temporary uses of variables. The slogan is:

Values are *taken* from the context, references are *borrowed* from the context.

Because flow typing treats the context as a state parameter, we can compile away our flow typing machinery by performing the state-passing explicitly. In this way, flow typing can be seen as a clean syntax for working with state in a linear type theory. More specifically, programs in our language can be translated into a suitable *linear lambda calculus* [5].

To make our theory concrete, we have implemented a lightweight, Haskell-like programming language with flow typing. We use this language for examples in the paper, and write code in a `teletype` font to emphasise that it is real, executable code rather than pseudocode. Our compiler uses the explicit state-passing translation to convert this language to Linear Haskell [6], without using any unsafe features to bypass the linearity checker.

The article itself makes the following contributions:

- We introduce the basic idea of flow typing by example, demonstrating how flow typing references work and how they can be compiled away (section 2);
- We formalise the idea by defining flow typing rules for a simple linear language (section 3) and then extend the language to handle references (section 4);
- We show how to translate flow-typed terms into linear lambda calculus terms (section 5), then show that this translation is semantics-preserving (section 6);
- Finally, we discuss our compiler and its extensions to the language (section 7), and conclude with an extended example to compare flow typing to other systems (section 8).

We discuss related work in section 9 and future work in section 10. Most of the article assumes only a basic knowledge of type systems and some intuition about linearity. We use a graphical notation (string diagrams) in section 6 to reason about our semantics, but our presentation does not assume experience with the underlying categorical formalisation of string diagrams. Our flow typing compiler and a set of example programs are freely available as supplementary material.

2 The Basic Idea

In this section we introduce the basic idea of flow typing. For example, in our language we can compute the length of a list using flow typing as follows:

```
length :: [a] &> Int
length &[]     = 0
length &(x:xs) = length &xs + 1
```

This definition looks similar to the traditional recursive definition in a Haskell-like language, but the additional flow typing machinery ensures that it is linear. We explain how to interpret the new syntax in this section, but the general intuition is that the operator & means *by reference*, and is read as 'ref', and the arrow &> is a type of functions taking their argument by reference, and is read as 'ref to'. This syntax was chosen to mirror the symbols used for references in languages like C, although here they refer to state-passing rather than pointers.

The flow typing definition for the `length` function is linear even though some variables are not used, in particular `x` in the second clause. Moreover, when our compiler translates this code into Linear Haskell, it emits precisely the more complicated definition from the introduction, including rebuilding the input list. How does this translation process work?

In flow typing, the typing context is a state parameter, and evaluating each expression manipulates the context in some way. To show how our compiler understands this code, we will traverse each expression in the definition and describe how it updates the context. For clarity, we will work with a desugared form of this program in which pattern matching is replaced by explicit case analysis:

```
length :: [a] &> Int
length = \&ys -> case &ys of
                   &[]     -> 0
                   &(x:xs) -> length &xs + 1
```

2.1 Calling by Reference

At the start of type checking the above definition, the typing context is empty. The compiler encounters the abstraction `\&ys -> ...`, which is an abstraction taking its argument *by reference*. Its type is `[a] &> Int`, which means essentially the same thing as `[a] -> (Int, [a])`.

Crucially, there is no type of references. Calling by reference here simply means that the value is returned to the caller after the function has been executed. This is implemented using explicit value-passing, so there are no actual pointers being used, but our syntax hides this internal plumbing. In particular, the type `a &> b` does not mean '`&a -> b`', which would amount to adding a type of references to the language; instead, it is just a different kind of arrow.

To interpret this &-abstraction in flow typing, we add `ys :: [a]` to the context now, and at the end of the abstraction, we expect that `ys :: [a]` is back in the context. Internally, we will return this new value of `ys` along with the usual return value of the abstraction.

2.2 Pattern-Matching by Reference

The next expression the compiler encounters is `case &ys of ...`, called a *pattern-match by reference*. The idea is that the argument `ys :: [a]` is temporarily removed from the context to be used for pattern-matching, and is put back at the end of the case expression. The compiler splits into two paths, one for each case. In each branch, the state of the context is manipulated differently, but the two paths must rejoin at the end of the case expression.

In the first branch, the variable `ys :: [a]` is removed from the context, then the expression `0` is evaluated, which does not change the context. Now that the compiler has reached the end of the branch, it tries to *reconstruct* the variable `ys`. Since we are in the branch where `ys` was the empty list, we simply rebuild `ys` in the context as a new copy of the empty list.

The second branch is slightly more complicated. Here, `ys` is removed from the context, but `x :: a` and `xs :: [a]` are both added. To evaluate the branch, in particular we need to evaluate `length &xs`. This is done by temporarily removing `xs` from the context, calling `length` on it, and then adding the returned list back into the context as `xs`. At the end of the branch, therefore, the two variables `x :: a` and `xs :: [a]` are back in the context, so we rebuild `ys` as `(x:xs)`. In particular, this removes `x` and `xs` from the context and adds `ys`. Therefore, both branches return integers and conclude with the same context `ys :: [a]`, so the pattern-match is accepted.

2.3 Finishing Up

Now that the body of the abstraction has been evaluated, the compiler encounters the end of the `\&ys -> ...` abstraction. A variable `ys :: [a]` is present in the current context, so we remove it and use it as the list returned from the abstraction. We now make a final check to ensure that the context is now empty, guaranteeing that we cannot discard any resources.

Once the state-passing described in this section has been unwound, we arrive at precisely the same linear definition for `length` as given in the introduction. This example illustrates how flow typing works as a natural syntax for stateful programming in a linear type theory. We could also consider more sophisticated implementations of flow typing, for instance a translation that implements our notion of references as pointers to improve efficiency. However, in this article we focus on the core translation to explicit state passing, as this shows that no complex language features are needed to implement flow typing. In the following sections, we will introduce flow typing more formally, and explain its relation to other linear languages.

3 Flow Typing Rules

This section is the start of the formal development of flow typing. We begin by introducing a simple form of flow typing as an alternative presentation of a linear lambda calculus. Then in section 4 we will show how to extend this language to add references.

3.1 Linear Type Systems

In conventional linear type systems, the typing judgements are of the form $\Gamma \vdash e : A$. Such a judgement means that the list of assumptions in the typing context Γ can be used by the expression e to produce a value of type A. For convenience, we will assume that the context Γ contains no duplicate variable names. The typing rules are restricted to guarantee that each variable is used exactly once. For instance, the variable rule is as follows:

$$\frac{}{x : A \vdash x : A}\ \textsc{Lin-Var}$$

Since this rule can only be applied when the context is a single variable, no resources can be discarded. Essentially all of the inference rules need to be slightly modified to align with this linearity constraint. As another example, the application rule is as follows, where we use the syntax '$A \multimap B$' (read 'A *wand* B') for the type of linear functions from A to B to emphasise the difference between it and the usual non-linear function type $A \to B$:

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : B}\ \textsc{Lin-}{\multimap}\textsc{-E}$$

To apply this rule, the context must be split into two parts, Γ_1 and Γ_2. This ensures that a variable cannot be used both in the function and in the argument of an application. For instance, the double application $f\ (f\ x)$ is not well-typed in this linear lambda calculus, because f is used twice.

In fig. 1, we expand the linear lambda calculus with rules for linear pair types $A \otimes B$, a unit type I, sum types $A \oplus B$, and an empty type which we write $\mathbf{f}$ (pronounced 'false'). We also add rules for some of the connectives unique to linear logic, namely the *additive product* $A \mathbin{\&} B$ ('A with B') and its unit $\mathbf{t}$ ('true'), as well as the exponential type $!A$ ('bang A'). The additive product is not to be confused with the reference operator &; they can be disambiguated by noting that the additive product is a binary operator and references are unary, but regardless, the two symbols will never appear close to each other in this article.

These extra connectives are not particularly important for this work, but we include them for completeness and to show that flow typing works with them. Intuitively, the additive product $A \mathbin{\&} B$ is a closure that can be invoked to produce either an A or a B, but is used up after being executed; the unit $\mathbf{t}$ represents a closure that can never be executed; and the type $!A$ represents a closure that can be invoked arbitrarily many times (including none at all) to produce values of type A.

$$\frac{}{x : A \vdash x : A}\,\textsc{Var} \qquad \frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2, x : A \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, x = e_1\, \mathrm{in}\, e_2) : B}\,\textsc{Let} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{true}(e) : \mathbf{t}}\,\mathbf{t}\text{-I}$$

$$\frac{\Gamma \vdash e : \mathbf{f}}{\Gamma \vdash \mathrm{false}_B(e) : B}\,\mathbf{f}\text{-E} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x\!:\!A.\, e) : A \multimap B}\,{\multimap}\text{-I}$$

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : B}\,{\multimap}\text{-E} \qquad \frac{}{\vdash * : I}\,I\text{-I} \qquad \frac{\Gamma_1 \vdash e_1 : I \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash (\mathrm{let} * = e_1\, \mathrm{in}\, e_2) : A}\,I\text{-E}$$

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : A \otimes B}\,{\otimes}\text{-I} \qquad \frac{\Gamma_1 \vdash e_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, (x, y) = e_1\, \mathrm{in}\, e_2) : C}\,{\otimes}\text{-E}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \langle e_1, e_2 \rangle : A \,\&\, B}\,\&\text{-I} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash \mathrm{fst}(e) : A}\,\&\text{-EL} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash \mathrm{snd}(e) : B}\,\&\text{-ER}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{inl}_{A \oplus B}(e) : A \oplus B}\,{\oplus}\text{-IL} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathrm{inr}_{A \oplus B}(e) : A \oplus B}\,{\oplus}\text{-IR}$$

$$\frac{\Gamma_1 \vdash e_1 : A \oplus B \qquad \Gamma_2, x : A \vdash e_2 : C \qquad \Gamma_2, y : B \vdash e_3 : C}{\Gamma_1, \Gamma_2 \vdash (\mathrm{case}\, e_1\, \mathrm{of}\, \mathrm{inl}(x) \rightarrow e_2 \mid \mathrm{inr}(y) \rightarrow e_3) : C}\,{\oplus}\text{-E}$$

$$\frac{\Gamma_1 \vdash e_1 : !A_1 \qquad \cdots \qquad \Gamma_n \vdash e_n : !A_n \qquad x_1 : !A_1, , x_n : !A_n \vdash e : B}{\Gamma_1, \ldots, \Gamma_n \vdash \mathrm{promote}\, e_1, \ldots, e_n\, \mathrm{for}\, x_1, \ldots, x_n\, \mathrm{in}\, e : !B}\,\textsc{Prom}$$

$$\frac{\Gamma_1 \vdash e_1 : !A \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathrm{discard}\, e_1\, \mathrm{in}\, e_2}\,\textsc{Disc} \qquad \frac{\Gamma_1 \vdash e_1 : !A \qquad \Gamma_2, x : !A, y : !A \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathrm{copy}\, e_1\, \mathrm{as}\, x, y\, \mathrm{in}\, e_2}\,\textsc{Copy}$$

$$\frac{\Gamma \vdash e : !A}{\Gamma \vdash \mathrm{derelict}\, e : A}\,\textsc{Der}$$

Fig. 1. Conventional presentation of linear lambda calculus

Remark. The phrase 'linear lambda calculus' has a few different meanings; we use it to mean the system with rules defined in fig. 1. This presentation of a linear lambda calculus was inspired by Bierman [7]. A more proof-theoretic introduction to this kind of system can be found in [4, 5]. ⋄

3.2 Flow Typing

In contrast to conventional linear type systems, *flow typing* judgements have the form $\Gamma_1 \vdash e : A \dashv \Gamma_2$. This judgement means that the context Γ_1 can be used by the expression e to produce a value of type A, and that the context that remains after evaluating e is Γ_2. This leftover context Γ_2 contains every variable that was not used by e, as well as variables that were merely borrowed in e and then put back into the context. As an example, the flow typing variable rule is:

$$\frac{}{\Gamma, x : A \vdash x : A \dashv \Gamma}\ \textsc{Flow-Var}$$

This rule states that if the context contains the binding $x : A$, we can remove it and yield a result of type A. The output context is exactly the same as the input context, except that the binding for x has been used up. Similarly, we can introduce a unit type I, with the following introduction rule that does not access or manipulate the context:

$$\frac{}{\Gamma \vdash * : I \dashv \Gamma}\ \textsc{Flow-}I\textsc{-I}$$

In more complicated expressions, the context is a state parameter, threaded through each subexpression to be evaluated. For example, the application rule is as follows:

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : A \dashv \Gamma_3}{\Gamma_1 \vdash e_1\ e_2 : B \dashv \Gamma_3}\ \textsc{Flow-}{\multimap}\textsc{-E}$$

Here, the two subexpressions e_1 and e_2 are evaluated sequentially, with the output context from e_1 being used as the input context for e_2. Note that the linear logic rule for application forces us to split the context into two parts to apply the rule, but the flow typing rule avoids this by instead stipulating an evaluation order for the subexpressions. The rule for pairing is similar, using the syntax $A \otimes B$ for the type of pairs of A and B:

$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (e_1, e_2) : A \otimes B \dashv \Gamma_3}\ \textsc{Flow-}{\otimes}\textsc{-I}$$

Example 1. Suppose we want to type check the double application $f\ (f\ x)$. Let Γ be the initial context $f : (A \multimap B), x : A$. The type derivation must begin as follows:

$$\frac{\Gamma \vdash f : A \multimap B \dashv x : A \qquad x : A \vdash (f\ x) : ? \dashv ?}{\Gamma \vdash (f\ (f\ x)) : ? \dashv ?}$$

We must now type check the subexpression $f\ x$ in a context that contains no variable f, which is impossible. This is the way that linearity is enforced in flow typing: once a variable is used, it is permanently removed from the context, so cannot be used again. ◇

3.3 Let Expressions

Compare the typing rules for 'let' expressions in our linear lambda calculus and in flow typing:

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2, x : A \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) : B}\ \textsc{Lin-Let} \qquad \frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) : B \dashv \Gamma_3}\ \textsc{Flow-Let}$$

At first, the flow typing rule looks like a simple restatement of the linear rule, but there is a crucial difference: we do not restrict the scope of x in the flow typing rule. This means that let-bound variables can escape their scope of definition. For example, the expression '$\mathrm{let}\, x = e \,\mathrm{in}\, *$' is valid using flow typing, and the output context has an additional binding for x. In section 5, we show how this scope-escaping behaviour is compiled away by our translation into a linear lambda calculus.

Example 2. Consider the expression '$\mathrm{let}\, y = x \,\mathrm{in}\, *$'. This has the following typing derivation:

$$\frac{\Gamma, x : A \vdash x : A \dashv \Gamma \qquad \Gamma, y : A \vdash * : I \dashv \Gamma, y : A}{\Gamma, x : A \vdash (\mathrm{let}\, y = x \,\mathrm{in}\, *) : I \dashv \Gamma, y : A}$$

Therefore, the expression '$\mathrm{let}\, y = x \,\mathrm{in}\, *$' simply *renames* x to y in the context. We can then consider the following larger expression:

$$e \;\coloneqq\; \mathrm{let} * = (\mathrm{let}\, y = x \,\mathrm{in}\, *) \,\mathrm{in}\, y$$

The use of y does not occur within the scope of its definition. Despite this, the expression above is well-typed, assuming the input context has the binding $x : A$. Indeed, we can derive:

$$\frac{\Gamma, x : A \vdash (\mathrm{let}\, y = x \,\mathrm{in}\, *) : I \dashv \Gamma, y : A \qquad \Gamma, y : A \vdash y : A \dashv \Gamma}{\Gamma, x : A \vdash e : A \dashv \Gamma}$$

So this expression is well-typed in the context $\Gamma, x : A$, and the output context is Γ. This example suggests that a more natural 'let' construct in flow typing might be a let *statement*, namely '$\mathrm{let}\, x = e$', declaring a variable x that can be used in later statements. We can then execute statements in sequence by using a sequencing operator '$e_1; e_2$', defined to be equivalent to '$\mathrm{let} * = e_1 \,\mathrm{in}\, e_2$'. ◇

3.4 Abstractions

The linear and flow typing rules for abstraction are as follows:

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x{:}A.\, e) : A \multimap B}\ \textsc{Lin-}{\multimap}\textsc{-I} \qquad\qquad \frac{\Gamma_1, x : A \vdash e : B \dashv}{\Gamma_1, \Gamma_2 \vdash (\lambda x{:}A.\, e) : A \multimap B \dashv \Gamma_2}\ \textsc{Flow-}{\multimap}\textsc{-I}$$

The linear rule passes the entire context into the abstraction to avoid discarding resources. However, the flow typing rule requires a context split: it allows the abstraction to capture the Γ_1 part of the environment, while preserving the Γ_2 part for later expressions to use. To ensure that resources are never discarded, we additionally assume that after evaluating e, the context is empty.

Remark. Consider the following alternative rule for abstraction.

$$\frac{\Gamma_1, x : A \vdash e : B \dashv \Gamma_2}{\Gamma_1 \vdash (\lambda x{:}A.\, e) : A \multimap B \dashv \Gamma_2}\ \textsc{Flow-}{\multimap}\textsc{-I}'$$

With this rule, the context is not split: it flows directly into the abstraction body and then out for later expressions to use. While this rule is simpler, it has the wrong computational interpretation. Consider the following expression, assuming that we have added integers to our language:

$$\lambda x{:}\mathit{Int}.\, (\mathrm{let}\, y = x + 1 \,\mathrm{in}\, *)$$

With the $\textsc{Flow-}{\multimap}\textsc{-I}'$ rule, this is type-correct. We can type check it as follows:

$$\frac{x : \mathit{Int} \vdash (\mathrm{let}\, y = x + 1 \,\mathrm{in}\, *) : I \dashv y : \mathit{Int}}{\vdash (\lambda x{:}\mathit{Int}.\, (\mathrm{let}\, y = x + 1 \,\mathrm{in}\, *)) : (\mathit{Int} \multimap I) \dashv y : \mathit{Int}}$$

Such a derivation seems to imply that the abstraction puts $y : \mathit{Int}$ in the context, even without having invoked the abstraction. But this does not make sense: the value of y depends on the value of x that has not yet been supplied to the closure. For this reason, we forbid abstractions from 'leaking' their variables to the outer scope, as achieved by the form of the $\textsc{Flow-}{\multimap}\textsc{-I}$ rule. ◇

3.5 Case Expressions

As another example, consider a case expression (writing $\oplus$ for the sum type):

$$\frac{\Gamma_1 \vdash e_1 : A \oplus B \qquad \Gamma_2, x : A \vdash e_2 : C \qquad \Gamma_2, y : B \vdash e_3 : C}{\Gamma_1, \Gamma_2 \vdash (\text{case}\, e_1\, \text{of}\, \text{inl}(x) \longrightarrow e_2 \mid \text{inr}(y) \longrightarrow e_3) : C}\ \textsc{Lin-}\oplus\textsc{-E}$$

$$\frac{\Gamma_1 \vdash e_1 : A \oplus B \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : C \dashv \Gamma_3 \qquad \Gamma_2, y : B \vdash e_3 : C \dashv \Gamma_3}{\Gamma_1 \vdash (\text{case}\, e_1\, \text{of}\, \text{inl}(x) \longrightarrow e_2 \mid \text{inr}(y) \longrightarrow e_3) : C \dashv \Gamma_3}\ \textsc{Flow-}\oplus\textsc{-E}$$

First, the linear rule is easy to interpret: it splits the context, using Γ_1 to evaluate the scrutinee and Γ_2 to evaluate each branch, ensuring that the result of each branch has the same type C. In contrast, the flow typing rule needs to check slightly more. Like the linear rule, it checks that the branches e_2 and e_3 yield the same type C, but it must also check that the two output contexts match, and uses this as the output context for the overall expression.

Example 3. Consider the following expression:

$$\begin{aligned} e \;\coloneqq\; & \text{case}\, x\, \text{of}\ \text{inl}(y) \longrightarrow (\text{inl}(y), 0) \\ & \qquad\quad \mid \text{inr}(z) \longrightarrow (\text{inr}(z), 1) \end{aligned}$$

The branches have the following derivations:

$$\frac{\cdots}{y : A \vdash (\text{inl}(y), 0) : (A \oplus B) \otimes \mathit{Int} \dashv} \qquad \frac{\cdots}{z : B \vdash (\text{inr}(z), 1) : (A \oplus B) \otimes \mathit{Int} \dashv}$$

Formally, we should add type annotations to inl(y) and inr(z) so that their type derivations are uniquely determined, as we do in the rules in fig. 2, but for clarity we suppress these annotations where possible. Note that the two output types and contexts match, so the overall expression e is type-correct: its type is $(A \oplus B) \otimes \mathit{Int}$ and the output context is empty. Now consider the following expression which swaps the variants in a sum type, leaving its result in the context:

$$\begin{aligned} e' \;\coloneqq\; & \text{case}\, x\, \text{of}\ \text{inl}(y) \longrightarrow (\text{let}\, x = \text{inr}(y)\, \text{in}\, *) \\ & \qquad\quad \mid \text{inr}(z) \longrightarrow (\text{let}\, x = \text{inl}(z)\, \text{in}\, *) \end{aligned}$$

The two branches have the following type derivations:

$$\frac{\cdots}{y : A \vdash (\text{let}\, x = \text{inr}(y)\, \text{in}\, *) : I \dashv x : B \oplus A} \qquad \frac{\cdots}{z : A \vdash (\text{let}\, x = \text{inl}(z)\, \text{in}\, *) : I \dashv x : B \oplus A}$$

Similarly, both branches have the same type and output context, so e' is type-correct.

$$\frac{\cdots}{x : A \oplus B \vdash e' : I \dashv x : B \oplus A}$$

◇

The full listing of rules, including the extra linear logic connectives, can be found in fig. 2. In the remainder of this section, we will establish some basic, metatheoretic results about our current flow typing system, showing how our language is related to the linear lambda calculus. Afterwards, in section 4, we will extend our language to add references.

Remark. We allow ourselves to use the structural rule of *exchange* to permute both input and output contexts. It is possible to bypass this extra rule by regarding contexts as multisets rather than lists; however, as this complicates some of the proofs later on, we will not do so in this article. ◇

$$\frac{}{\Gamma, x : A \vdash x : A \dashv \Gamma}\ \textsc{Var} \qquad \frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) : B \dashv \Gamma_3}\ \textsc{Let}$$

$$\frac{\Gamma_1 \vdash e : A \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{true}(e) : \mathbf{t} \dashv \Gamma_2}\ \mathbf{t}\text{-I} \qquad \frac{\Gamma_1 \vdash e : \mathbf{f} \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{false}_B(e) : B \dashv \Gamma_2}\ \mathbf{f}\text{-E}$$

$$\frac{\Gamma_2, x : A \vdash e : B \dashv}{\Gamma_1, \Gamma_2 \vdash (\lambda x : A.\, e) : A \multimap B \dashv \Gamma_1}\ \multimap\text{-I} \qquad \frac{\Gamma_1 \vdash e_1 : A \multimap B \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : A \dashv \Gamma_3}{\Gamma_1 \vdash e_1\ e_2 : B \dashv \Gamma_3}\ \multimap\text{-E}$$

$$\frac{}{\Gamma \vdash * : I \dashv \Gamma}\ I\text{-I} \qquad \frac{\Gamma_1 \vdash e_1 : I \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : A \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let} * = e_1 \,\mathrm{in}\, e_2) : A \dashv \Gamma_3}\ I\text{-E}$$

$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (e_1, e_2) : A \otimes B \dashv \Gamma_3}\ \otimes\text{-I}$$

$$\frac{\Gamma_1 \vdash e_1 : A \otimes B \dashv \Gamma_2 \qquad \Gamma_2, x : A, y : B \vdash e_2 : C \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let}\, (x, y) = e_1 \,\mathrm{in}\, e_2) : C \dashv \Gamma_3}\ \otimes\text{-E}$$

$$\frac{\Gamma_2 \vdash e_1 : A \dashv \qquad \Gamma_2 \vdash e_2 : B \dashv}{\Gamma_1, \Gamma_2 \vdash \langle e_1, e_2 \rangle : A \,\&\, B \dashv \Gamma_1}\ \&\text{-I} \qquad \frac{\Gamma_1 \vdash e : A \,\&\, B \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{fst}(e) : A \dashv \Gamma_2}\ \&\text{-EL} \qquad \frac{\Gamma_1 \vdash e : A \,\&\, B \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{snd}(e) : B \dashv \Gamma_2}\ \&\text{-ER}$$

$$\frac{\Gamma_1 \vdash e : A \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{inl}_{A \oplus B}(e) : A \oplus B \dashv \Gamma_2}\ \oplus\text{-IL} \qquad \frac{\Gamma_1 \vdash e : B \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{inr}_{A \oplus B}(e) : A \oplus B \dashv \Gamma_2}\ \oplus\text{-IR}$$

$$\frac{\Gamma_1 \vdash e_1 : A \oplus B \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : C \dashv \Gamma_3 \qquad \Gamma_2, y : B \vdash e_3 : C \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{case}\, e_1 \,\mathrm{of}\, \mathrm{inl}(x) \longrightarrow e_2 \mid \mathrm{inr}(y) \longrightarrow e_3) : C \dashv \Gamma_3}\ \oplus\text{-E}$$

$$\frac{\Gamma_1 \vdash e_1 : !A_1 \dashv \Gamma_2 \qquad \cdots \qquad \Gamma_n \vdash e_n : !A_n \dashv \Gamma_{n+1} \qquad x_1 : !A_1, \ldots, x_n : !A_n \vdash e : B \dashv}{\Gamma_1 \vdash \mathrm{promote}\, e_1, \ldots, e_n \,\mathrm{for}\, x_1, \ldots, x_n \,\mathrm{in}\, e : !B \dashv \Gamma_{n+1}}\ \textsc{Prom}$$

$$\frac{\Gamma_1 \vdash e_1 : !A \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash \mathrm{discard}\, e_1 \,\mathrm{in}\, e_2 : B \dashv \Gamma_3}\ \textsc{Disc}$$

$$\frac{\Gamma_1 \vdash e_1 : !A \dashv \Gamma_2 \qquad \Gamma_2, x : !A, y : !A \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash \mathrm{copy}\, e_1 \,\mathrm{as}\, x, y \,\mathrm{in}\, e_2 : B \dashv \Gamma_3}\ \textsc{Copy} \qquad \frac{\Gamma_1 \vdash e : !A \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{derelict}\, e : A \dashv \Gamma_2}\ \textsc{Der}$$

Fig. 2. Flow typing rules for linear lambda calculus

Remark. Unlike in the presentation of linear logic in fig. 1, flow typing requires a context split in only two places, namely, the introduction rules for implication $\multimap$ and additive product &. The computational meaning of these splits is that these connectives are interpreted as closures, and so they represent places where computation 'forks': some calculation happens now and some is suspended for later. The other connectives that are interpreted as closures, namely **t** and $!A$, are given rules that avoid context splits by construction. Concretely, the introduction rule for **t** is explicitly given the data it captures as its argument; this is required in order to have uniqueness of derivations. Similarly, the introduction ('promotion') rule for $!A$ has an explicit list of the variables that the closure can use; this decision calls back to Benton et al. [4, 5] who chose this to ensure that the derivations in their linear lambda calculus were closed under substitution. ⋄

3.6 Linear Terms are Flow Typed Terms

It is straightforward to show that the current flow typing rules are a slight generalisation of the usual linear rules. Formally, we have the following theorem:

THEOREM 3.1. *If* $\Gamma \vdash e : A$*, then* $\Gamma \vdash e : A \dashv$ *(with empty output context).*

Remark. The reverse direction does not hold, because let-bound variables can escape their scope of definition in flow typing, but cannot in the linear lambda calculus. ⋄

The proof makes use of a simple lemma:

LEMMA 3.2 (FRAME RULE). *We can temporarily remove variables from the context while performing a derivation. That is, the following rule is admissible (can be proven valid using our current rules):*

$$\frac{\Gamma_1 \vdash e : A \dashv \Gamma_2 \qquad x \text{ does not appear in } \Gamma_1, \Gamma_2, e}{\Gamma_1, x : B \vdash e : A \dashv \Gamma_2, x : B}\ \textsc{Frame}$$

Remark. The name of this lemma echoes the frame rule from *separation logic* [24, 26]. The frame rule is normally presented in terms of *Hoare triples*, assertions of the form $\{P\}\ S\ \{Q\}$, where P is a *precondition* that we assume holds before execution, S is a *statement* to be executed, and Q is the *postcondition* that holds after execution. The usual statement is that a suitably 'separate' assumption R can always be added to both P and Q while preserving validity of the assertion. The similarity of this rule to our flow typing frame rule indicates that we can think of flow typing judgements themselves as Hoare triples: the input context Γ_1 is the precondition required for the derivation, an expression e is a statement producing a value, and Γ_2 is the postcondition. ⋄

3.7 Uniqueness of Derivations

We can also prove that typing derivations are essentially unique if they exist.

THEOREM 3.3. *An expression e has at most one type and output context (up to permutation), given a fixed input context* Γ_1*.*

$$(\Gamma_1 \vdash e : A \dashv \Gamma_2) \;\wedge\; (\Gamma_1 \vdash e : A' \dashv \Gamma_2') \quad\Longrightarrow\quad A = A' \;\wedge\; \Gamma_2 \text{ is a permutation of } \Gamma_2'$$

PROOF. By structural induction on expressions, generalising over types and contexts (which are understood to be identified up to permutation throughout this proof). Using the fact that typing is syntax-directed, most cases are trivial. The only nontrivial cases are when the context splits, which occurs only in ⊸-I and &-I. Suppose that we are comparing the following two occurrences of ⊸-I:

$$\frac{\Gamma_2, x : A \vdash e : B \dashv}{\Gamma_1, \Gamma_2 \vdash (\lambda x \!:\! A.\, e) : A \multimap B \dashv \Gamma_1} \qquad\qquad \frac{\Gamma_2', x : A \vdash e : B' \dashv}{\Gamma_1', \Gamma_2' \vdash (\lambda x \!:\! A.\, e) : A \multimap B' \dashv \Gamma_1'}$$

We apply the frame rule (lemma 3.2) to the two assumptions to obtain:

$$\Gamma_1, \Gamma_2, x : A \vdash e : B \dashv \Gamma_1 \qquad\qquad \Gamma_1', \Gamma_2', x : A \vdash e : B' \dashv \Gamma_1'$$

But Γ_1, Γ_2 and Γ_1', Γ_2' are equal up to permutation, so the inductive hypothesis now applies, giving $B = B'$ and $\Gamma_1 = \Gamma_1'$ up to permutation. We can thus conclude also that $\Gamma_2 = \Gamma_2'$ up to permutation. The same trick can be used to prove the inductive step for &-I, thus giving the result. □

This result also provides an easy proof of the following:

Corollary 3.4. *A linear lambda calculus expression e has at most one type in any context Γ.*

$$(\Gamma \vdash e : A) \wedge (\Gamma \vdash e : A') \implies A = A'$$

Proof. Both assumptions can be translated by theorem 3.1 to flow typing derivations, for which theorem 3.3 provides a proof that $A = A'$. □

We consider it interesting that this diversion through flow typing allows us to avoid reasoning about the ubiquitous context splits in the linear lambda calculus, and instead reduces our focus to the two context splits in the flow typing rules.

Remark. Benton et al. [4, 5, theorem 1] claim that if the expression e and type A are fixed, then the context Γ is uniquely determined. Unfortunately, due to presence of weakening (the Disc rule), that result holds neither in their system nor in ours. As a minimal example, in the expression $(\text{discard}\, x \,\text{in}\, y) : B$, the context Γ can be set to $x : !A, y : B$ for any choice of A. ◇

4 References

In this section, we add a form of references to our flow typing language. Recall that manipulating a variable 'by reference' in flow typing means to temporarily take the variable out of the typing context and put it back later; in particular, there is no *type* of references. The two basic ingredients we need to add are pattern-matching by reference and calling by reference. We crucially rely on flow typing judgements to describe the semantics of these constructs.

4.1 Pattern-Matching by Reference

To begin, consider the flow typing rule for pattern-matching on a pair:

$$\frac{\Gamma_1 \vdash e_1 : A \otimes B \dashv \Gamma_2 \qquad \Gamma_2, x : A, y : B \vdash e_2 : C \dashv \Gamma_3}{\Gamma_1 \vdash (\text{let}\,(x, y) = e_1 \,\text{in}\, e_2) : C \dashv \Gamma_3}\ \text{Flow-}\otimes\text{-E}$$

Here, the expression e_1 is evaluated and then split apart into $x : A$ and $y : B$. We would like a similar expression for pattern-matching on a pair by reference, leaving the pair in the context after the body of the let-binding has finished. To do this, we first need to restrict this rule to act only on variables, so that we have a name to use to put the pair back in the context at the end:

$$\frac{\Gamma_1, x : A, y : B \vdash e : C \dashv \Gamma_2}{\Gamma_1, z : A \otimes B \vdash (\text{let}\,(x, y) = z \,\text{in}\, e) : C \dashv \Gamma_2}$$

To make this rule into a pattern-match *by reference*, we need a way to reconstruct the value of the pair after the body has been executed. The simplest way to do this is just to assume that the variables x and y are still in the context, as follows:

$$\frac{\Gamma_1, x : A, y : B \vdash e : C \dashv \Gamma_2, x : A, y : B}{\Gamma_1, z : A \otimes B \vdash (\text{let}\,\&(x, y) = \&z \,\text{in}\, e) : C \dashv \Gamma_2, z : A \otimes B}\ \&\otimes\text{-E}$$

The new value of z in the output context will be precisely (x, y).

Example 4. Consider the following expression that uses pattern-matching by reference to increment the first element of a pair $z : \mathit{Int} \otimes \mathit{Int}$ 'in place' (leaving its result in the context):

$$e \;\coloneqq\; \mathrm{let}\,\&(x, y) = \&z\,\mathrm{in}\,(\mathrm{let}\,x = x + 1\,\mathrm{in}\,*)$$

The inner let-expression has the following typing derivation in the context $x : \mathit{Int}, y : \mathit{Int}$:

$$\dfrac{\dfrac{\cdots}{x : \mathit{Int}, y : \mathit{Int} \vdash x + 1 : \mathit{Int} \dashv y : \mathit{Int}} \qquad y : \mathit{Int}, x : \mathit{Int} \vdash * : I \dashv y : \mathit{Int}, x : \mathit{Int}}{x : \mathit{Int}, y : \mathit{Int} \vdash (\mathrm{let}\,x = x + 1\,\mathrm{in}\,*) \dashv y : \mathit{Int}, x : \mathit{Int}}$$

Since the output context contains the integers x and y, the overall expression e is well-typed, and its output context is $z : \mathit{Int} \otimes \mathit{Int}$. ◇

We can make a similar 'by reference' form of the flow typing rule for the case expression. Recall that the 'by value' flow typing rule for the case expression is as follows:

$$\dfrac{\Gamma_1 \vdash e_1 : A \oplus B \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : C \dashv \Gamma_3 \qquad \Gamma_2, y : B \vdash e_3 : C \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{case}\,e_1\,\mathrm{of}\,\mathrm{inl}(x) \longrightarrow e_2 \mid \mathrm{inr}(y) \longrightarrow e_3) : C \dashv \Gamma_3}\ \textsc{Flow-}\oplus\textsc{-E}$$

Restricting to variables, we get:

$$\dfrac{\Gamma_1, x : A \vdash e_1 : C \dashv \Gamma_2 \qquad \Gamma_1, y : B \vdash e_2 : C \dashv \Gamma_2}{\Gamma_1, z : A \oplus B \vdash (\mathrm{case}\,z\,\mathrm{of}\,\mathrm{inl}(x) \longrightarrow e_1 \mid \mathrm{inr}(y) \longrightarrow e_2) : C \dashv \Gamma_2}$$

We need to reconstruct the sum type after each branch has finished. In the first branch, we need to assume that x is still in the context, allowing us to reconstruct z as $\mathrm{inl}(x)$. Similarly, in the right branch we assume that y is still in the context. The rule thus has the following form:

$$\dfrac{\Gamma_1, x : A \vdash e_1 : C \dashv \Gamma_2, x : A \qquad \Gamma_1, y : B \vdash e_2 : C \dashv \Gamma_2, y : B}{\Gamma_1, z : A \oplus B \vdash (\mathrm{case}\,\&z\,\mathrm{of}\,\&\mathrm{inl}(x) \longrightarrow e_1 \mid \&\mathrm{inr}(y) \longrightarrow e_2) : C \dashv \Gamma_2, z : A \oplus B}\ \&\oplus\text{-E}$$

Note that the remainders of the two output contexts, written here as Γ_2, must still match.

Example 5. We can rewrite example 3 in a more concise way using references. Consider the following expression, which returns 1 if the value x is of the form $\mathrm{inr}(-)$:

$$\begin{aligned} e \;\coloneqq\; & \mathrm{case}\,\&x\,\mathrm{of}\,\&\mathrm{inl}(y) \longrightarrow 0 \\ & \qquad\qquad \mid \&\mathrm{inr}(z) \longrightarrow 1 \end{aligned}$$

It is easy to see that this is well-typed:

$$\dfrac{y : A \vdash 0 : \mathit{Int} \dashv y : A \qquad z : B \vdash 1 : \mathit{Int} \dashv z : B}{x : A \oplus B \vdash e : \mathit{Int} \dashv x : A \oplus B}$$

The output context contains x, indicating it was not used up and can hence be used again later. ◇

Our language has four data type constructors: $\otimes$ and $\oplus$, which we have already seen, as well as the empty type $\mathbf{f}$ and the unit type I. The flow typing elimination rule for $\mathbf{f}$ is:

$$\dfrac{\Gamma_1 \vdash e : \mathbf{f} \dashv \Gamma_2}{\Gamma_1 \vdash \mathrm{false}_B(e) : B \dashv \Gamma_2}\ \textsc{Flow-}\mathbf{f}\textsc{-E}$$

Following the pattern, this rule has the following reference form:

$$\dfrac{}{\Gamma_1, x : \mathbf{f} \vdash \mathrm{false}_B(\&x) : B \dashv \Gamma_1, x : \mathbf{f}}\ \&\mathbf{f}\text{-E}$$

This rule allows us to use a variable $x : \mathbf{f}$ to produce new data of an arbitrary type without destroying this capability in the process. In section 5, we will implement this rule using $\mathrm{false}_{(B\otimes\mathbf{f})}(x)$, which returns the desired value of type B as well as a new value for $x : \mathbf{f}$. In comparison, the reference rule for the unit type has no practical use, because its only behaviour is to temporarily remove a variable $x : I$ from the context, a feature already covered by the *frame rule* (lemma 3.2):

$$\frac{\Gamma_1 \vdash e : A \dashv \Gamma_2}{\Gamma_1, x : I \vdash (\mathrm{let}\,\&{*} = \&x\,\mathrm{in}\,e) : A \dashv \Gamma_2, x : I}\ \&I\text{-E}$$

Remark. These examples can be composed to give a pattern-match by reference rule for all algebraic data types. Our compiler implements this strategy in order to allow pattern-matching by reference on recursively defined data types such as lists and trees. ◇

4.2 Calling by Reference

The final ingredient that we need to add to flow typing is a notion of function that takes its argument by reference. Recall that the lambda abstraction typing rule is as follows:

$$\frac{\Gamma_2, x : A \vdash e : B \dashv}{\Gamma_1, \Gamma_2 \vdash (\lambda x\!:\!A.\,e) : A \multimap B \dashv \Gamma_1}\ \textsc{Flow-}\multimap\text{-I}$$

The machinery for *ref-functions* will be implemented in a similar way to the pattern-matching by reference constructions we have made so far. We design the rules for our new '$\overset{\&}{\multimap}$' connective in such a way that $A \overset{\&}{\multimap} B$ means essentially the same thing as $A \multimap B \otimes A$. Concretely, a *lambda-ref* abstraction puts its parameter in the context at the start, and once the body of the function is over, it expects that its parameter is still in the context. Its typing rule is as follows:

$$\frac{\Gamma_2, x : A \vdash e : B \dashv x : A}{\Gamma_1, \Gamma_2 \vdash (\lambda\&x\!:\!A.\,e) : A \overset{\&}{\multimap} B \dashv \Gamma_1}\ \overset{\&}{\multimap}\text{-I}$$

Example 6. The identity-by-reference function is written:

$$(\lambda\&x\!:\!A.\,*) : A \overset{\&}{\multimap} I$$

This expression takes x by reference, does nothing to it, and returns the unit $*$. Even though the variable x was never used, it is picked up by the ref-function machinery and returned implicitly, thereby ensuring that linearity is preserved. ◇

We also need an eliminator for such function types. It has the following form:

$$\frac{\Gamma_1 \vdash e : A \overset{\&}{\multimap} B \dashv \Gamma_2, x : A}{\Gamma_1 \vdash e\ \&x : B \dashv \Gamma_2, x : A}\ \overset{\&}{\multimap}\text{-E}$$

This modified function application syntax takes a variable out of the context, evaluates the function, then returns the variable back to the context.

Example 7. We can apply a function $f : A \overset{\&}{\multimap} I$ to the first element of a pair $z : A \otimes B$ in place, using the following expression:

$$e \ :=\ \mathrm{let}\,\&(x, y) = \&z\,\mathrm{in}\,f\ \&x$$

This pattern-matches on the pair by reference to put the variable x in the context, then applies f to it, and puts the new value of x back in the context. At the end of the pattern-match, the variables x and y are taken out of the context and used to reconstruct $z = (x, y)$. ◇

Example 8. Functions $A \multimap A$ correspond precisely to functions $A \overset{\&}{\multimap} I$. Concretely, we have the following maps to convert between the two representations:

$$\begin{aligned} P &:= \lambda f : (A \multimap A).\ \lambda \& x : A.\ (\text{let}\, x = f\ x\, \text{in}\, *) : (A \multimap A) \multimap (A \overset{\&}{\multimap} I) \\ Q &:= \lambda f : (A \overset{\&}{\multimap} I).\ \lambda x : A.\ (\text{let}\, * = f\ \& x\, \text{in}\, x)\ \ : (A \overset{\&}{\multimap} I) \multimap (A \multimap A) \end{aligned}$$

More generally, there is an isomorphism between $A \overset{\&}{\multimap} B$ and $A \multimap B \otimes A$. In practice, the two forms have their own advantages and disadvantages, so it is useful to have these combinators to convert between the two styles of function definition. Interestingly, this shows a way in which our notion of references is more powerful than Rust's mutable references, because the Rust language has no safe way to define the P combinator. However, there is an *unsafe* (not borrow-checked) implementation of this function in the popular `replace_with` crate [22]. ◇

Example 9. Ref-functions cannot be 'curried'. More precisely, we do not have the bijection:

$$((A \otimes B) \overset{\&}{\multimap} C)\ \cong\ (A \overset{\&}{\multimap} (B \overset{\&}{\multimap} C))$$

One reason for the lack of such a bijection is that a function $f : A \overset{\&}{\multimap} (A \overset{\&}{\multimap} B)$ can be called with the same argument twice, as in $f\ \& x\ \& x$. This is because the first invocation returns x to the context before the second invocation occurs. However, a function $g : (A \otimes A) \overset{\&}{\multimap} B$ requires both of its arguments at the same time, so cannot be evaluated with two instances of the same argument. Therefore, the lack of currying is a fundamental restriction imposed by linearity, rather than a limitation of flow typing. Despite this, we can always 'uncurry' nested ref-functions, as follows:

$$R\ :=\ \lambda f : (A \overset{\&}{\multimap} (B \overset{\&}{\multimap} C)).\ \lambda \& x : (A \otimes B).\ \text{let}\, \&(a, b) = \& x\, \text{in}\, f\ \& a\ \& b$$

The absence of currying might seem like a limitation, but we will see how to work around this in section 7 when we discuss our compiler for a more general flow typing language. ◇

5 Translation to Linear Lambda Calculus

In this section, we prove that there is a translation from flow typed terms to linear lambda calculus terms. We achieve this by encoding manipulations of the context using explicit state passing. This will establish one of our main goals, which is that flow typing does not rely on any complicated language features to implement. More precisely, we define the translation:

$$(\Gamma_1 \vdash e : A \dashv \Gamma_2) \quad \longrightarrow \quad (\Gamma_1 \vdash \hat{e} : A \otimes \Gamma_2^\circ)$$

Here, Γ_2° is a type corresponding to a 'packed' version of the output context Γ_2, defined recursively as follows (where $\diamond$ denotes the empty context):

$$(\diamond)^\circ = I; \quad (x : A, \Gamma)^\circ = A \otimes \Gamma^\circ$$

Writing '$\hat{e}$' here is a slight abuse of notation, since the translation is actually defined by recursion on typing judgements, but it should be clear at all points which typing judgement for e is meant when this syntax is used. The full list of translation rules is given in a supplementary appendix, but we describe the general technique here with specific examples.

To do this, we first introduce some *packing* and *unpacking* expressions that allow us to convert a context Γ to and from its packed form Γ°. Let 'pack Γ' be the expression given recursively by:

$$\text{pack}\,(\diamond) = *; \quad \text{pack}\,(x : A, \Gamma) = (x, \text{pack}\,\Gamma)$$

Thus we have the linear lambda calculus judgement:

$$\frac{}{\Gamma \vdash \text{pack}\,\Gamma : \Gamma^\circ}\ \textsc{Pack}$$

Example 10. The flow typing variable rule is translated as follows:

$$\frac{}{\Gamma, x : A \vdash x : A \dashv \Gamma}\ \textsc{Var} \quad \mapsto \quad (x, \mathrm{pack}\,\Gamma)$$

That is, the translation of the variable x in the context $\Gamma, x : A$ is given by a pair that stores both the result of the expression x and the packed output context Γ. This translation has the following linear lambda calculus typing judgement:

$$\frac{}{\Gamma, x : A \vdash (x, \mathrm{pack}\,\Gamma) : A \otimes \Gamma^\circ} \qquad \diamond$$

To compose these translations, we also need a way to unpack contexts. Concretely, we recursively define an unpacking operation as follows:

$$\begin{aligned} \mathrm{unpack}'\,\gamma\,\mathrm{as}\,\diamond\,\mathrm{in}\,e &:= \mathrm{let}\,{*} = \gamma\,\mathrm{in}\,e \\ \mathrm{unpack}'\,\gamma\,\mathrm{as}\,(x : A, \Gamma)\,\mathrm{in}\,e &:= \mathrm{let}\,(x, \delta) = \gamma\,\mathrm{in}\,(\mathrm{unpack}'\,\delta\,\mathrm{as}\,\Gamma\,\mathrm{in}\,e) \end{aligned}$$

We then obtain the following rule, which gives us a simple way to use a packed context:

$$\frac{\Gamma_1 \vdash \gamma : \Gamma^\circ \qquad \Gamma, \Gamma_2 \vdash e : A}{\Gamma_1, \Gamma_2 \vdash \mathrm{unpack}'\,\gamma\,\mathrm{as}\,\Gamma\,\mathrm{in}\,e : A}\ \textsc{Unpack}'$$

Because we are generally working with pairs of a value and a packed context, we also define a version of 'unpack' that is more convenient for our use case:

$$\mathrm{unpack}\,e_1\,\mathrm{as}\,x, \Gamma\,\mathrm{in}\,e_2 := \mathrm{let}\,(x, \gamma) = e_1\,\mathrm{in}\,(\mathrm{unpack}'\,\gamma\,\mathrm{as}\,\Gamma\,\mathrm{in}\,e_2)$$

This has the following rule:

$$\frac{\Gamma_1 \vdash e_1 : A \otimes \Gamma^\circ \qquad x : A, \Gamma, \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathrm{unpack}\,e_1\,\mathrm{as}\,x, \Gamma\,\mathrm{in}\,e_2 : B}\ \textsc{Unpack}$$

Example 11. Consider the flow typing rule for let expressions:

$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let}\,x = e_1\,\mathrm{in}\,e_2) : B \dashv \Gamma_3}\ \textsc{Flow-Let}$$

By recursion on judgements, we may assume that we already have translations:

$$\Gamma_1 \vdash \hat{e}_1 : A \otimes \Gamma_2^\circ \qquad \Gamma_2, x : A \vdash \hat{e}_2 : B \otimes \Gamma_3^\circ$$

This allows us to write the following translation of the entire let expression:

$$\Gamma_1 \vdash (\mathrm{unpack}\,\hat{e}_1\,\mathrm{as}\,x, \Gamma_2\,\mathrm{in}\,\hat{e}_2) : B \otimes \Gamma_3^\circ$$

This translation explains why let-bound variables are allowed to escape their scope in flow typing: such a variable would be stored in Γ_3°, which the translated expression returns. Later, an expression will unpack Γ_3° back into its context form Γ_3, at which point this variable can be used. $\diamond$

Example 12. Many of the translation rules follow a straightforward pattern: unpack each argument in turn, apply the relevant linear lambda calculus operation, then pack up the rest of the context. For instance, the rules for application and pairing are translated as follows:

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : A \dashv \Gamma_3}{\Gamma_1 \vdash e_1\ e_2 : B \dashv \Gamma_3}\ \textsc{Flow-}{\multimap}\textsc{-E}$$

$$\mapsto \text{unpack}\ \hat{e}_1\ \text{as}\ x, \Gamma_2\ \text{in unpack}\ \hat{e}_2\ \text{as}\ y, \Gamma_3\ \text{in}\ (x\ y, \text{pack}\ \Gamma_3)$$

$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (e_1, e_2) : A \otimes B \dashv \Gamma_3}\ \textsc{Flow-}{\otimes}\textsc{-I}$$

$$\mapsto \text{unpack}\ \hat{e}_1\ \text{as}\ x, \Gamma_2\ \text{in unpack}\ \hat{e}_2\ \text{as}\ y, \Gamma_3\ \text{in}\ ((x, y), \text{pack}\ \Gamma_3)$$ ◇

Example 13. Recall that we allow both the input and output contexts to be freely permuted, by means of a structural rule called the *exchange* rule. Hence, we need to provide a way to interpret the exchange rule under this translation. Concretely, if Γ_1' is a permutation of Γ_1 and Γ_2' is a permutation of Γ_2, we need to define a mapping of derivations:

$$(\Gamma_1 \vdash e : A \otimes \Gamma_2) \quad \to \quad (\Gamma_1' \vdash \tilde{e} : A \otimes \Gamma_2')$$

Because the linear lambda calculus has the exchange rule for its (input) contexts, the permutation of Γ_1 presents no issues, and we do not need to change the form of e. For the output context, however, we need to perform the exchange manually using a pack-unpack pair:

$$\tilde{e} \ \coloneqq\ \text{unpack}\ e\ \text{as}\ x, \Gamma_2\ \text{in}\ (x, \text{pack}\ \Gamma_2')$$

Here, the exchange happens implicitly in the Pack rule:

$$\frac{}{\Gamma_2 \vdash \text{pack}\ \Gamma_2' : (\Gamma_2')^\circ}$$

This is a valid derivation because we can exchange the Γ_2 (which now takes the position of an input context in the linear lambda calculus) into Γ_2'. ◇

The rest of the expressions that came from the linear lambda calculus can be translated in a similar manner. The full list of translation rules can be found in the appendix. We conclude this section by discussing the rules for the parts of the language defined in section 4 relating to references.

Example 14. Consider the rule for pattern-matching by reference on a pair:

$$\frac{\Gamma_1, x : A, y : B \vdash e : C \dashv \Gamma_2, x : A, y : B}{\Gamma_1, z : A \otimes B \vdash (\text{let}\ \&(x, y) = \&z\ \text{in}\ e) : C \dashv \Gamma_2, z : A \otimes B}\ \&{\otimes}\text{-E}$$

We encode the behaviour of this rule by first using a normal pattern match, then reconstructing the value of z at the end. The expression translates as follows:

$$\begin{array}{l} \text{let}\ (x, y) = z \\ \quad \text{in unpack}\ \hat{e}\ \text{as}\ t, (\Gamma_2, x : A, y : B) \\ \quad\quad \text{in let}\ z = (x, y) \\ \quad\quad\quad \text{in}\ (t, \text{pack}\ (\Gamma_2, z : A \otimes B)) \end{array}$$

This is a type-correct linear lambda calculus term. We define the translation of the rule for pattern-matching by reference on a sum type in a similar way. Recall that the rule is as follows:

$$\frac{\Gamma_1, x : A \vdash e_1 : C \dashv \Gamma_2, x : A \qquad \Gamma_1, y : B \vdash e_2 : C \dashv \Gamma_2, y : B}{\Gamma_1, z : A \oplus B \vdash (\text{case}\ \&z\ \text{of}\ \&\text{inl}(x) \to e_1 \mid \&\text{inr}(y) \to e_2) : C \dashv \Gamma_2, z : A \oplus B}\ \&{\oplus}\text{-E}$$

It is translated to:

$$\begin{aligned}&\text{case}\ z\ \text{of}\ \text{inl}(x) \longrightarrow \text{unpack}\ \hat{e}_1\ \text{as}\ t, (\Gamma_2, x : A)\ \text{in}\ \text{let}\ z = \text{inl}(x)\ \text{in}\ (t, \text{pack}\ (\Gamma_2, z : A \oplus B))\\ &\qquad\quad \mid \text{inr}(y) \longrightarrow \text{unpack}\ \hat{e}_2\ \text{as}\ t, (\Gamma_2, y : B)\ \text{in}\ \text{let}\ z = \text{inr}(y)\ \text{in}\ (t, \text{pack}\ (\Gamma_2, z : A \oplus B))\end{aligned}$$

The remaining pattern-matching by reference rules are translated as follows:

$$\frac{}{\Gamma_1, x : \mathbf{f} \vdash \text{false}_B(\&x) : B \dashv \Gamma_1, x : \mathbf{f}}\ \&\mathbf{f}\text{-E}$$
$$\mapsto \text{let}\ (t, x) = \text{false}_{(B\otimes\mathbf{f})}(x)\ \text{in}\ (t, \text{pack}\ (\Gamma_1, x : \mathbf{f}))$$

$$\frac{\Gamma_1 \vdash e : A \dashv \Gamma_2}{\Gamma_1, x : I \vdash (\text{let}\ \&* = \&x\ \text{in}\ e) : A \dashv \Gamma_2, x : I}\ \&I\text{-E}$$
$$\mapsto \text{unpack}\ \hat{e}\ \text{as}\ t, \Gamma_2\ \text{in}\ (t, \text{pack}\ (\Gamma_2, x : I))$$ ◇

To translate the ref-functions defined in section 4.2, we need to define what happens to the type $A \overset{\&}{\multimap} B$. As alluded to above, we translate this to the linear lambda calculus type $A \multimap B \otimes A$.

Example 15. We can now define the following translations for ref-functions:

$$\frac{\Gamma_2, x : A \vdash e : B \dashv x : A}{\Gamma_1, \Gamma_2 \vdash (\lambda\&x{:}A.\, e) : A \overset{\&}{\multimap} B \dashv \Gamma_1}\ \overset{\&}{\multimap}\text{-I}$$
$$\mapsto ((\lambda x{:}A.\ (\text{unpack}\ \hat{e}\ \text{as}\ y, (x : A)\ \text{in}\ (y, x))), \text{pack}\ \Gamma_1)$$

$$\frac{\Gamma_1 \vdash e : A \overset{\&}{\multimap} B \dashv \Gamma_2, x : A}{\Gamma_1 \vdash e\ \&x : B \dashv \Gamma_2, x : A}\ \overset{\&}{\multimap}\text{-E}$$
$$\mapsto \text{unpack}\ \hat{e}\ \text{as}\ y, (\Gamma_2, x : A)\ \text{in}\ \text{let}\ (z, x) = y\ x\ \text{in}\ (z, \text{pack}\ (\Gamma_2, x : A))$$

These translations encode how the value of x is retrieved from the context at the end of the function, and how the argument passed to the function is returned to the context. ◇

Therefore, we have established:

THEOREM 5.1. *If* $\Gamma_1 \vdash e : A \dashv \Gamma_2$, *then* $\Gamma_1 \vdash \hat{e} : A \otimes \Gamma_2^\circ$.

We have implemented this translation in our compiler, which we discuss in section 7.

6 Semantics

Once we have defined a semantics for the linear lambda calculus, we get a semantics for flow typing for free via theorem 5.1: the semantics of a flow typing term e can be *defined* as the semantics of $\hat{e}$. In particular, we have a simple semantics for our notion of references.

However, there is a key unanswered question: if e is already a term in the linear lambda calculus, we can regard it as a flow typed term by theorem 3.1, and then convert to a new linear expression $\hat{e}$. Do e and $\hat{e}$ have the same semantics? This is not just a theoretical question, it has practical significance; if this is true, it shows that an optimising compiler can compile reference-free flow typing programs as efficiently as linear programs. Fortunately, this is true: more precisely, we will show in this section that $\hat{e}$ is semantically equivalent to $(e, *)$.

To get this result, we define a semantics in terms of *string diagrams* [13], which we will give an informal introduction to in this section. This is for two reasons: first, string diagrams provide a convenient graphical notation to describe our technique. Secondly, and more importantly, this semantics has a richer set of equalities than a simple β-reduction relation would have. These extra equalities, called *commuting conversions*, are required for some of the inductive steps in our proofs.

A flow typing compiler can implement some of these equalities as rewrite rules, allowing it to simplify away some of the explicit state passing involved in the translation in theorem 5.1. We will describe the rewrite rules carried out by our compiler in section 7.

Example 16. A simple β-reduction relation is not sufficient to get the desired conclusion above. As an example, consider the expression:

$$e \;\coloneqq\; (\mathrm{case}\, x \,\mathrm{of}\, \mathrm{inl}(y) \to 1 \mid \mathrm{inr}(z) \to 2)$$

Regarding this as a flow typing term and applying the translation from section 5, we obtain:

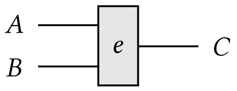
$$\hat{e} \xrightarrow{\beta} (\mathrm{case}\, x \,\mathrm{of}\, \mathrm{inl}(y) \to (1, *) \mid \mathrm{inr}(z) \to (2, *)) \overset{\beta}{\not\to} ((\mathrm{case}\, x \,\mathrm{of}\, \mathrm{inl}(y) \to 1 \mid \mathrm{inr}(z) \to 2), *)$$

This cannot be β-reduced any further since x is a variable. Despite this, the string diagram semantic interpretations of $\hat{e}$ and $(e, *)$ agree. ◇

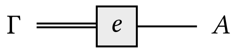
Remark. The string diagrams and associated proofs that we describe below can all be formalised in terms of a categorical semantics; see [12, 35] for examples on how this is achieved. However, in this article, we will work only with an informal understanding of string diagrams, and defer our technical comments to a remark at the end of this section. ◇

6.1 Informal Introduction to String Diagrams

String diagrams are a convenient tool to write down expressions in linear type theories. Expressions are written as boxes, with their inputs on the left and and their output on the right. For instance:

$x : A, y : B \vdash e : C$ is written as 

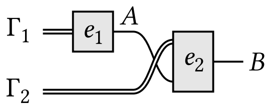
Each 'string' has a fixed type, and should be thought of as a wire carrying information from the left of the diagram to the right. In general, an expression might have many input wires, one for each variable in the context. We draw these lists of wires bundled together, as follows:

$\Gamma \vdash e : A$ is written as 

Note that the way that diagrams are structured enforces linearity: the fact that a wire must go from its origin to exactly one endpoint ensures that values cannot be duplicated or deleted.

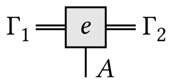
Diagrams can be composed by joining up wires of matching types. For instance, we can write the linear rule for let expressions as follows:

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2, x : A \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) : B}\ \textsc{Let} \quad \mapsto \quad (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) =$$



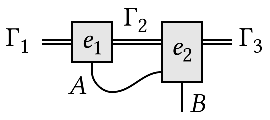
Here, the context is comprised of two bundles. The first bundle is provided to the expression e_1, and its result is passed alongside the other bundle to e_2 to compute the result.

String diagrams are not only useful for drawing linear lambda calculus terms, but also for flow typing terms. To do this, we write an expression e as a box, where the input context is on the left, the output context is on the right, and the result of the expression points below the box.

$\Gamma_1 \vdash e : A \dashv \Gamma_2 \quad \mapsto$ 

This choice of graphical notation emphasises the fact that the context flows through expressions. Here, the rule for let expressions is depicted as follows:

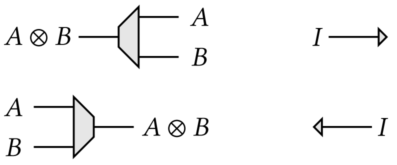
$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vdash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vdash (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) : B \dashv \Gamma_3}\ \textsc{Let} \quad \mapsto \quad (\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) =$$



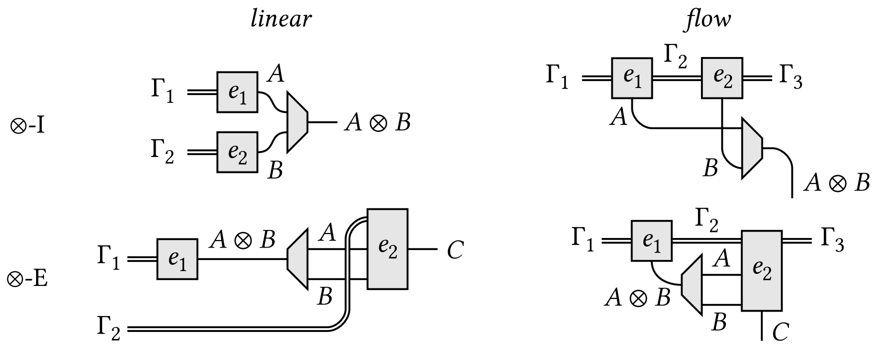
The diagram for this rule clearly shows the sequential nature of computation in flow typing: the context indeed flows directly through each expression to be evaluated in turn.

6.2 Diagrams for Connectives

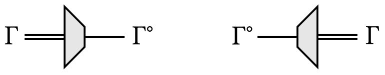
To write the rules for the other connectives, we need operations to construct and deconstruct pairs, the unit type, sum types, and so on. For instance, we define operators for ⊗ and I as follows:



The connectors for I, for example, allow us to freely create and delete objects of type I. We can now write the rules for pairing as follows:



We also overload this notation for packing and unpacking contexts:

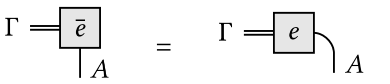


The full list of rules for both the linear lambda calculus and the flow typing calculus can be found in our technical appendix, attached as supplementary material to this article.

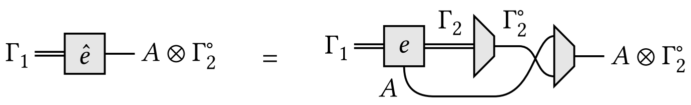
6.3 Equivalence

Our equivalence results, which we will now state, concern equalities of string diagrams. We regard two string diagrams as equal if they have the same categorical interpretation. In practice, this means that we can slide wires and boxes around in the diagrams, including crossing wires over, without changing the meaning of the diagram. We can also perform some simplifications, such as cancelling adjacent packing and unpacking operations.

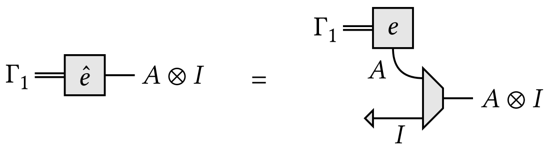
Theorem 6.1 (translation to flow typing). *Let $\Gamma \vdash e : A$, and write $\overline{e}$ for the judgement $\Gamma \vdash e : A \dashv$ as defined in theorem 3.1. Then,*



Theorem 6.2 (translation to linear lambda calculus). *Let $\Gamma_1 \vdash e : A \dashv \Gamma_2$, and write $\hat{e}$ for the judgement $\Gamma_1 \vdash \hat{e} : A \otimes \Gamma_2^\circ$ as defined in theorem 5.1. Then,*



Corollary 6.3. *Let e be a linear lambda calculus term. Regarding it as a flow typing term and translating it back into a linear lambda calculus term, we obtain:*



That is, $\hat{e}$ is semantically equivalent to $(e, *)$.

Proof of both theorems. By induction on typing derivations, making use of some simple lemmas that establish the semantics of the frame rule and the packing operations. □

Remark. To complete this proof, we made the following assumptions on the category $\mathscr{C}$ that our string diagrams are constructed in:

- for tensor $\otimes$, we assume $\mathscr{C}$ is a symmetric monoidal category;
- for linear implication $\multimap$, we assume each of the functors $A \otimes (-)$ has a right adjoint $A \multimap (-)$, making $\mathscr{C}$ a *monoidal closed* category;
- for additive disjunction $\oplus$ and unit $\mathbf{f}$, we assume $\mathscr{C}$ has finite coproducts;
- for additive conjunction & and unit $\mathbf{t}$, we assume $\mathscr{C}$ has finite products;
- for the exponential $!(-)$, we assume that $\mathscr{C}$ is equipped with a comonad $\mathbb{G}$, together with designated morphisms of the following types:

$$\text{wkg} : GA \longrightarrow I; \quad \text{ctr} : GA \longrightarrow GA \otimes GA; \quad \text{join}_0 : I \longrightarrow GI; \quad \text{join}_2 : GA \otimes GB \longrightarrow G(A \otimes B)$$

These assumptions are a weakening of the definition of a *linear category* from [7]. We note that for the additives & and $\oplus$, we really need products and coproducts; the *weak* products and coproducts used by Benton et al. [4] do not suffice for our purposes. This assumption is needed in order to appeal to extensionality when proving the inductive steps for these connectives. However, it turns out that the proofs require no laws on the morphisms associated with the comonad $\mathbb{G}$. ◇

7 The Compiler

We have implemented the translation of section 5 in a compiler, which is available as supplementary material for the article. This compiler translates our Haskell-like flow typing language into Linear Haskell, which can then be run using any modern version of GHC. Crucially, because the compiler uses the process described in section 5 to generate its output, Haskell can verify that the generated code is linear, and so we do not need to use unsafe features to bypass the linearity checker.

The compiler itself is written in approximately 2500 lines of Haskell, including a custom parser and error reporting system. The type checker consists of 600 lines of code, and the translation to a linear language is implemented in 500 lines.

We have tested our compiler on 6 kB of examples which are available as supplementary material. These examples translate to 43 kB of Linear Haskell code, after formatting using Ormolu [19]. After running a simple optimiser which we discuss below, the output is reduced to 13 kB.

7.1 Generalising to Irrefutable Patterns

Our compiler supports a slightly more general language than the one discussed so far, and in this section we will discuss this generalisation. As noted in example 9, ref-functions cannot be curried, as there is not a bijection of the form:

$$((A \otimes B) \overset{\&}{\multimap} C) \;\cong\; (A \overset{\&}{\multimap} (B \overset{\&}{\multimap} C))$$

As discussed in that example, the inability to curry ref-functions is a fundamental restriction imposed by linearity, not a limitation of flow typing.

This fact means that if a function needs to take multiple arguments by reference, these arguments must be packaged into a tuple. However, the rules defined in section 4.2 stipulate that the argument to such a function must be a single variable; in particular, the syntax $f\ \&(x, y)$ is invalid. To address this, our compiler allows an arbitrary *irrefutable pattern* to follow the & symbol in a pattern-match by reference or call-by-reference expression. This makes the above syntax valid, as well as the corresponding syntax for nested tuples and the unit type. This generalisation is simply syntax sugar, as we can make transformations such as the following:

$$\begin{aligned} f\ \&(x, y) \;\mapsto\; & \mathrm{let}\, z = (x, y) \\ & \mathrm{in}\,\mathrm{let}\, t = f\ \&z \\ & \quad \mathrm{in}\,\mathrm{let}\,(x, y) = z\,\mathrm{in}\, t \end{aligned}$$

Any irrefutable pattern can be used in place of the pair (x, y), and the compiler will emit corresponding code. In fact, we additionally permit integer and boolean literals, which are translated in the following way, using the integer literal 2 as an example:

$$\begin{aligned} f\ \&2 \;\mapsto\; & \mathrm{let}\, z = 2 \\ & \mathrm{in}\,\mathrm{let}\, t = f\ \&z \\ & \quad \mathrm{in}\,\mathrm{let}\,\_ = z\,\mathrm{in}\, t \end{aligned}$$

This makes use of the fact that we are free to discard integers, as they contain no resources.

7.2 Optimisation

The translation from section 5 emits far from optimal code, because it systematically inserts ‘pack’ and ‘unpack’ operations, many of which are redundant and can be eliminated. Therefore, we apply a small set of optimisations to the emitted Linear Haskell code, which also makes the output more readable. The three optimisations are case-of-case, case-of-known constructor, and let-inlining, all of which are local rewrite rules [25]. Because our type system is linear, let-inlining will never cause work duplication as noted in [25], so we apply this transformation unconditionally.

In practice, these three simple transformations are enough to greatly simplify the output code. When applied to our examples, the size of the output is reduced from 43 kB of Linear Haskell code down to 13 kB. The reason for this is that the proofs in section 6 only rarely appeal to universal properties. Instead, most of the work in the proofs came from cancelling ‘unpack’ and ‘pack’ pairs, which case-of-known constructor achieves, as well as moving blocks around a string diagram, which is achieved by case-of-case and let-inlining.

8 Extended Example

In this section, we compare flow typing to the approach to linearity that is adopted in two other languages, namely Linear Haskell [6] and Rust [21]. In particular, we present an implementation of a binary search algorithm in each of the three languages. The input data is a (sorted) binary tree, the structure of which is given by the following Haskell datatype:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

8.1 Flow Typing

One implementation of binary search using flow typing is as follows:

```
search :: Int -> Tree Int &> Bool
search x &Leaf = dsc x; False
search x &(Node y lhs rhs) =
  if cpy &x < cpy &y      then search x &lhs
  else if cpy &x > cpy &y then search x &rhs
  else dsc x; True
```

Here, `cpy :: Int &> Int` and `dsc :: Int -> ()` are built-in functions which copy and discard an integer respectively. The notation `a; b` abbreviates `let () = a in b`, which allows the expression `a` to be executed purely for its effect on the context.

In the above implementation, the input tree is passed in by reference (section 4), and we use pattern-match by reference to determine whether it is a leaf or a node. Crucially, we do not need to rebuild the entire tree after traversing it, because this is handled automatically by pattern-matching by reference. In particular, the branch that recursively searches the left-hand side of the tree does not need to mention the right-hand side at all, and vice versa. This also helps to reduce the likelihood of programming errors by confusing the two sides of the tree.

Remark. We are going to compare flow typing with Linear Haskell and with Rust, both of which have a mechanism to use certain 'unrestricted' variables from the context more than once. In Linear Haskell, individual variables in the context can be marked as unrestricted, and in Rust, some data types are marked as `Copy`, which essentially makes all variables of those types unrestricted. Such a generalisation of flow typing is beyond the scope of this article, but would let us omit the usages of the `cpy` and `dsc` functions in this example, further simplifying the code. ◇

8.2 Linear Haskell

We now compare this code to the equivalent Linear Haskell, using the new linear let bindings introduced in GHC 9.10.1:

```
search :: Int -> Tree (Ur Int) %1 -> (Bool, Tree (Ur Int))
search x Leaf = (False, Leaf)
search x (Node (Ur y) lhs rhs) =
  if x < y then
    let !(r, t) = search x lhs in (r, Node (Ur y) t rhs)
  else if x > y then
    let !(r, t) = search x rhs in (r, Node (Ur y) lhs t)
  else (True, Node (Ur y) lhs rhs)
```

In this code, the syntax '`a %1 -> b`' denotes the type of arrows from `a` to `b` that use their argument exactly once, whereas '`a -> b`' denotes the type of functions from `a` to `b` that have *unrestricted* use of their argument (that is, these are usual nonlinear Haskell functions). Additionally, note that the strict pattern `!(r, t)` was used instead of the lazy pattern `(r, t)`; this is because lazy patterns are not allowed in Haskell's linear let-bindings. Finally, the type constructor `Ur` defines a type of unrestricted values: a linear value of type `Ur a` is the same as an unrestricted value of type `a`. Formally, it is given as the following GADT:

```
data Ur a where
  Ur :: a -> Ur a    -- unrestricted function type
```

Some of the bureaucracy of `Ur` can be alleviated using a library such as `linear-base` [28], which in particular defines an isomorphism between `Int` and `Ur Int`.

In the Linear Haskell code, the search function outputs both the boolean result and a reconstructed copy of the input tree, as required to preserve linearity. This makes the code more difficult to read and write. Importantly, the above code could not have been significantly simplified by putting it inside a monad. We could not use a state monad, for example, as the argument for the recursive call is not the same as the input parameter. This example reflects a practicality issue with linear languages without the ability to locally escape linearity (such as with references): functional programs often look up information from recursive data structures, but linearity often requires that the structures be destroyed and rebuilt in order to do so.

8.3 Rust

Finally, we compare our flow typing code with the equivalent Rust code. Rust has a type `&T` of safe, *borrow-checked* references, which we will use in our implementation.

```
fn search(x: i32, t: &Tree<i32>) -> bool {
    match t {
        Tree::Leaf => false,
        Tree::Node(y, lhs, rhs) => {
            if x < *y { search(x, &lhs) }
            else if x > *y { search(x, &rhs) }
            else { true }
        }
    }
}
```

This code is much cleaner, and more closely resembles the flow typing version. The code works because integers and borrows have unrestricted usage, and so can be used as many times as desired. However, the underlying mechanics used to make these references work are more complicated than the surface-level code makes them appear. In reality, the function signature is elaborated as follows, where `'a` is a *lifetime variable*, a tag used by the borrow checker:

```
fn search<'a>(x: i32, t: &'a Tree<i32>) -> bool { ... }
```

In particular, this function is polymorphic over the lifetime of the borrow of `t`. The extensive use of lifetime polymorphism in Rust can make it harder to use, especially in higher-order cases.

Rust also has a notion of pattern-matching by reference (an inspiration for our approach in section 4), which is used above to pattern match on `t`. Here, `y` has the type `&'a i32`, the type of references with lifetime `'a` to a 32-bit integer. Similarly, `lhs` and `rhs` are given the type `&'a Tree<i32>`. The borrow checker uses these tags to ensure none of these borrows are retained past the end of the lifetime `'a`. Since lifetimes passed to Rust functions must always outlive the function scope, the borrow checker decides that the rules of borrowing are satisfied, and accepts the program.

This reliance on a borrow checker is a disadvantage of the Rust-style approach. First, it presents a significant implementation challenge to any designer wishing to add this notion of borrowing to a new language, and it increases the surface area of a compiler that must be trusted by its users. Additionally, the rules for what a borrow checker should allow are complex and are still not completely determined; see [16, 32] for recent attempts to finalise them.

8.4 Conclusion

This example shows how flow typing can be used as a simple yet practical syntax for programming with recursive data structures in a linear type theory. In comparison with Linear Haskell and Rust, we believe that our system sits in a new part of the design space, and has the potential to improve the readability and understanding of linear code.

9 Related Work

In this section, we survey related work and position the notion of flow typing within the literature. There are three main areas that we will discuss: functional presentations of imperative phenomena such as references, the interplay between references and linear types, and ways in which linearity has been made more practical and production-ready.

9.1 Functional Presentations of References

Perhaps the most well-known example of modelling imperative phenomena in a functional setting is by Swierstra and Altenkirch [31], who demonstrate how the 'awkward squad' of I/O, mutable state, and concurrency can be modelled using free monads. More specifically for references, the most widely-known functional presentation is the study of *lenses*, sometimes simply called 'functional references', which were first introduced in [11]. A lens provides a way to view and update a component part of a larger object. Historically, lenses were primarily used in nonlinear languages, but recent work has shown how to extend these ideas to a linear setting [27]. The central connection to our work is that a variable in a context can be seen as a lens, where the act of using a variable by reference updates its value in the context. In this way, part of our work can be seen as a simple syntax for constructing and using certain forms of lenses.

Lenses are not the only functional approach to references. For example, Kagawa [18] introduced compositional references, language-level constructs that allow the state monad ST to work not only on entire values, but also on component parts of values. Our references can be described in a similar way, allowing us to statefully manipulate component parts of the context.

As another example, the Aeneas project of Ho and Protzenko [14] describes a functional semantics for Rust, making use of a translation to a functional language. In the same way as our translation in section 5, in this project mutability is encoded using as state-passing, making use of the fact that mutable references cannot be duplicated. However, our goals are quite different: their result was a tool that enables formal theorem proving about Rust programs, whereas our goal is to make programming with linearity more lightweight and practical.

9.2 References with Linear Types

The interactions between references and linear types have been studied in various contexts. The most well-known example (albeit with a form of affine types, not true linear types) is Rust-style borrowing [21]. In this system, values can be temporarily 'borrowed', allowing them to be used by reference. Through the use of *lifetimes* annotated by the programmer, a borrow checker is able to determine a class of programs for which all references are guaranteed to point to living objects, among other guarantees. Formal developments of the Rust language include [2, 14, 17, 34], and the borrow checker itself has been developed in [16, 32].

An early example of Rust-style borrowing can be found in Wadler's original article on linear types [33]. In particular, Wadler's *let*! construct allows temporary nonlinear access to linear values, using a strict type-based analysis to ensure that references to the linear value cannot be held once the argument of the *let*! has been fully evaluated. This idea was refined by Odersky [23] to introduce 'observer' variables, which behave more similarly to Rust's immutable borrows. Similarly, Fahndrich and DeLine [9] use guarded types and capabilities in a similar way to Rust-style lifetimes, ensuring that references are discarded by the time that the referent is used again.

Another approach to combining references with linear types is to separate permissions from data. This is the strategy used in the L^3 language of Ahmed et al. [1], in which the (linear) capability to access or mutate a pointer is separated from the (nonlinear) pointer itself. This allows for shared mutable references and cyclic data structures, but without giving up semantic properties such as termination. Notably, Ahmed et al. describe the explicit threading of capabilities through a program as "too painful to contemplate"; a flow typing version of L^3 might alleviate this problem as it would allow capabilities to be passed by reference rather than being explicitly threaded.

9.3 Practical Linearity

This article is one of many attempts to make linear types practical for use in real-world programming. One well-known practical implementation of linear types is in Linear Haskell [6], which is a compiler extension distributed with GHC, and is therefore easily available to all Haskell programmers. One benefit of Linear Haskell is its implementation of multiplicity polymorphism, allowing programs to be written polymorphically over linear and unrestricted arguments.

However, Linear Haskell provides no safe mechanism to temporarily escape linearity, such as with references or flow typing, which means that objects often need to be threaded explicitly through safe Linear Haskell code. To alleviate this, a significant amount of such code, such as in `linear-base` [28] (a standard library for Linear Haskell), makes use of the '`Unsafe.toLinear`' combinator, marking a function as linear without compile-time checks. More recently, linear constraints have been introduced by Spiwack et al. [29]. Linear constraints can be filled in automatically using a typeclass solver, and can therefore reduce the syntactic overhead of explicit threading.

Another approach to practical linearity is the use of quantitative type theory [3], in which variables in the context are annotated with a multiplicity, typically one of 0 (erased at runtime), 1 (linear), or ω (unrestricted). Simple algebraic rules describe how multiplicities of variables are altered as contexts pass through expression constructors. This has been implemented in Idris 2 [8], a dependently-typed functional programming language.

The Cyclone language, a type-safe dialect of C, uses a similar technique to the L^3 language discussed above, making use of linear capabilities to encode dynamic regions and unique pointers [10]. To manage the bureaucracy of threading capabilities through a program, they provide a mechanism to temporarily 'open' a capability, allowing it to be used easily.

10 Conclusions and Future Work

In this article, we introduced a new type theoretic tool called *flow typing*, based on the idea of treating the typing context as a state parameter. We demonstrated that flow typing supports a simple notion of references that avoids the use of a borrow checker or complicated type theory. Flow typing provides a language that is easier to use than a linear lambda calculus, and we have demonstrated how it can be used to simplify various linear programs. However, since all of its machinery can be compiled away using explicit state passing, we retain the same powerful reasoning principles that conventional linear programming languages enjoy. We have implemented this theory in a programming language with a compiler that outputs Linear Haskell.

There are a number of possible directions for further work. First of all, it would be useful to develop a GHC plugin for flow typing, allowing our techniques to be used directly in Linear Haskell code. This would enable Linear Haskell programmers to write programs in a more natural style. Secondly, we note that our 'by reference' rules currently only operate on values. We saw in section 7 that these rules can be extended to work on irrefutable patterns, but it is possible to go further. One possible generalisation is to treat a reference as a *lens* (e.g. [27]) over part of the context. From this perspective, flow typing can be seen as a way to implement a language suitable for bidirectional programming. Additionally, the current translation of flow typing into Linear Haskell compiles references away into explicit value manipulation, so in particular, they do not have the runtime efficiency usually associated with references. It would be an interesting future direction of research to implement our references as real pointers in hardware.

And finally, while this article has focused on the use of flow typing to describe types of expressions, it may also be more generally applicable, such as with permissions or capabilities. It would be interesting to explore generalisations such as these in more detail, and see if our results about 'compiling away' references can be applied to these settings.

Acknowledgments

This work was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*, for which funding is gratefully acknowledged.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

[2] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 88–108.

[3] Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*. Association for Computing Machinery, 56–65. https://doi.org/10.1145/3209108.3209189

[4] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. 1992. *Term Assignment for Intuitionistic Linear Logic*. Technical Report. Technical Report 262, Computer Laboratory, University of Cambridge.

[5] Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. 1993. A Term Calculus for Intuitionistic Linear Logic. In *Typed Lambda Calculi and Applications*, Gerhard Goos, Juris Hartmanis, Marc Bezem, and Jan Friso Groote (Eds.). Vol. 664. Springer Berlin Heidelberg, Berlin, Heidelberg, 75–90. https://doi.org/10.1007/BFb0037099

[6] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[7] G.M. Bierman. 1994. *On Intuitionistic Linear Logic*. Technical Report UCAM-CL-TR-346. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-346

[8] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. https://arxiv.org/abs/2104.00480

[9] Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. *SIGPLAN Not.* 37, 5 (May 2002), 13–24. https://doi.org/10.1145/543552.512532

[10] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21.

[11] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem . *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 65 pages. https://doi.org/10.1145/1232420.1232424

[12] Dan Ghica and Fabio Zanasi. 2024. String Diagrams for λ-calculi and Functional Computation. arXiv:2305.18945 [cs.LO] https://arxiv.org/abs/2305.18945

[13] Ralf Hinze and Dan Marsden. 2023. *Introducing String Diagrams*. Cambridge University Press, Cambridge, England.

[14] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 711–741. https://doi.org/10.1145/3547647

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. *SIGPLAN Not.* 43, 1 (Jan. 2008), 273–284. https://doi.org/10.1145/1328897.1328472

[16] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371109

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–34. https://doi.org/10.1145/3158154

[18] Koji Kagawa. 1997. Compositional References for Stateful Functional Programming. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ACM, Amsterdam The Netherlands, 217–226. https://doi.org/10.1145/258948.258969

[19] Mark Karpov et al. 2025. ormolu: A Formatter for Haskell Source Code. https://github.com/tweag/ormolu

[20] Danielle Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 1040–1070. https://doi.org/10.1145/3649848

[21] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. ACM, Portland Oregon USA, 103–104. https://doi.org/10.1145/2663171.2663188

[22] Alec Mocatta et al. 2025. replace_with: Temporarily Take Ownership of a Value at a Mutable Location, and Replace It with a New Value Based on the Old One . https://github.com/alecmocatta/replace_with

[23] Martin Odersky. 1992. Observers for Linear Types. In *ESOP '92*, Bernd Krieg-Brückner (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 390–407.

[24] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *International Workshop on Computer Science Logic*. Springer, 1–19.

[25] Simon L. Peyton Jones. 1996. Compiling Haskell by Program Transformation: A Report from the Trenches. In *Programming Languages and Systems — ESOP '96*, Hanne Riis Nielson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–44.

[26] John C Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th annual IEEE symposium on logic in computer science*. IEEE, 55–74.

[27] Mitchell Riley. 2018. Categories of Optics. https://doi.org/10.48550/arXiv.1809.00738 arXiv:1809.00738 [math]

[28] Arnaud Spiwack et al. 2025. linear-base: Standard Library for Linear Types. https://github.com/tweag/linear-base

[29] Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (Aug. 2022), 28 pages. https://doi.org/10.1145/3547626

[30] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[31] Wouter Swierstra and Thorsten Altenkirch. 2007. Beauty in the Beast. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1291201.1291206

[32] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. https://doi.org/10.1145/3735592

[33] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*, Vol. 3. Citeseer, 5.

[34] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The Essence of Rust. https://doi.org/10.48550/ARXIV.1903.00982

[35] Paul Wilson, Dan Ghica, and Fabio Zanasi. 2024. String Diagrams for Strictification and Coherence. *Logical Methods in Computer Science* Volume 20, Issue 4 (Oct. 2024), 13982. https://doi.org/10.46298/lmcs-20(4:8)2024

[36] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and Programming Challenges of Rust: A Mixed-Methods Study. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1269–1281. https://doi.org/10.1145/3510003.3510164