# The Best of Both Worlds

J. Garrett Morris
The University of Edinburgh
Garrett.Morris@ed.ac.uk

## Abstract

We present a linear functional calculus with both the safety guarantees expressible with linear types and the rich language of combinators and composition provided by functional programming. Unlike previous combinations of linear typing and functional programming, we compromise neither the linear side (for example, our linear values are first-class citizens of the language) nor the functional side (for example, we do not require duplicate definitions of compositions for linear and unrestricted functions). To do so, we must generalize abstraction and application to encompass both linear and unrestricted functions. We capture the typing of the generalized typing constructs with a novel use of qualified types. Our system maintains the metatheoretic properties of the theory of qualified types, including principal types and decidable type inference. Finally, we give a formal basis for our claims of expressiveness, by showing that evaluation respects linearity, and that our language is a conservative extension of existing functional calculi.

## 1. Introduction

Integers have a pleasing consistency: values do not become more or less integers over the course of a computation. The same is not true for file handles: we can no longer expect to read from or write to a file handle after it has been closed. Traditional functional type systems, like the logic they resemble, are good for integers (i.e., unchanging beliefs), but less good for file handles (i.e., temporary ones). If our type systems are to help in the latter case, we need ones with a different logical character.

One approach is suggested by Girard's linear logic [10], which requires that each hypothesis be used exactly once in the course of a proof. Intuitively, linear propositions are finite resources, which can neither be duplicated nor discarded, rather than arbitrary truth values, available as often, or as rarely, as needed. Linear type systems adopt the same approach to variables: each bound variable must be used exactly once in the body of its binder. Such type systems have been used to reasoning about resource usage and concurrency. For example, they have been used to assure safe manipulation of state [1, 12], regulate access to shared resources [3, 6], and type interacting concurrent processes [4, 28, 30]. Each of these examples uses the restrictions on reuse and discard introduced by linearity to assure safety invariants. Simultaneously, several general purpose linear functional languages have been proposed, including those of Wadler [29], Mazurak et al. [20], and Tov and Pucella [27]. However, attempts to adapt functional programming idioms and abstractions to these calculi is thwarted by the interplay of linear and intuitionistic types. This paper proposes a novel combination of linear and qualified types that provides the safety of linear types without losing the expressiveness of functional programming.

We identify three challenges introduced by the interaction of linear types and functional programming. As an example, consider the $K$ combinator, defined as $\lambda x.\lambda y.x$. We begin with its arguments: argument $x$ is used once, and so can take on values of any type. Argument $y$ is discarded, and so can only take on values of unrestricted type. This illustrates the first challenge: we must distinguish between quantification over arbitrary type and quantification over unrestricted types. Next, consider the application $K\,V$, giving a new function $\lambda y.V$. Whether we can reuse this function depends on $V$. If $V$ is a Boolean or integer value, for example, there is no danger in reusing $\lambda y.V$. On the other hand, if $V$ is a file handle or capability, then reuse of the function would imply reuse of $V$, and should be prevented. This illustrates the second challenge: we must distinguish between linear and unrestricted functions, which distinction is determined by the environment captured by each function. In particular, there is no a priori assignment of linearity to the subterms of $K$ that accounts for its application to both linear and unrestricted values. Finally, consider the composition function $\circ$, defined as $\lambda f.\lambda g.\lambda x.f\,(g\,x)$. We know that we must be able to apply $f$ to things, and that $f$ is used linearly; however, these constraints are satisfied by both linear and unrestricted functions. So, the final challenge is that we must generalize the typing of application to range over the possible types of function.

These problems have been addressed in previous work, although their overlaps have not. Quantification over unrestricted values can be expressed either using kinds [20] or type classes [8]; either approach extends naturally to account for pairs and sums. Many existing systems use subtyping to account for application, either implicitly [8, 20] or explicitly [9, 27]. Tov and Pucella [27] introduce a notion of relative arrow qualifiers, sufficient to express the typing of $K$, but at the cost of significant complexity in their type system. The interplay of these disparate mechanisms has not been fully explored. For example, we do not believe that any of the existing systems can express the desired typing of composition, nor do they support complete type inference.

We propose a new, uniform approach to addressing the challenges of integrating linear types and functional programming, based on the theory of qualified types [15]. Rather than invent new type system features, we present a language design based on a novel combination of two existing, well-studied type systems. To demonstrate the features of our design, we return to the $K$ combinator, to which we give the type $(\mathsf{Un}\ u, t \geq f) \Rightarrow t \to u \xrightarrow{f} t$. First, we observed that the second argument (here typed by $u$) must be unrestricted; this is captured by the predicate $\mathsf{Un}\ u$. Second, we observed that the partial application $K\,V$ must be linear if $V$ is linear. We

capture this through the use of two predicates, one that identifies functions and another that specifies relative linearity. The predicate $\mathsf{Fun}\, f$ is satisfied only when $f$ is a function type; we write $t \xrightarrow{f} u$ to denote the type $f\, t\, u$ under the predicate $\mathsf{Fun}\, f$. The predicate $t \geq f$ is satisfied when $t$ supports more structural rules (i.e., duplication and discard) than $f$. Thus, in the typing of $K$, if $t$ is linear, then $f$ must be linear; alternatively, if $t$ is unrestricted, than $f$ can be either linear or unrestricted. We would make a similar use of the $\mathsf{Fun}$ predicate to account for the arguments of the composition operator.

Formally, we capture our approach in the design of a core linear calculus, which we call a Qualified Linear Language (Quill). Quill is a linear variant of Jones's calculus OML, extended with Haskell-like first-class polymorphism [14], and with entailment rules for the $\mathsf{Un}$, $\mathsf{Fun}$, and $\geq$ predicates. We preserve the metatheoretic properties of OML, particularly principal types and decidable type inference (without requiring the programmer to provide type or linearity annotations). We show that our system is a conservative extension of (non-linear) OML; concretely, this means that we can view our approach as giving linear refinements of existing functional languages and idioms, rather than replacing them entirely. Finally, we give a natural (big-step) semantics for Quill and show that evaluation respects linearity.

In summary, we contribute the following:

- The design and motivation of Quill, including examples of Quill's application to prototypical uses of linear types (binary session types) and higher-order functional programming (monads) (§3).
- A formal account of the Quill type system and its relationship to OML, including a sound and complete type inference algorithm (§4).
- A linearity-aware semantics of Quill, and a proof that values of linear type are neither duplicated nor discarded during evaluation (§5).
- A discussion of further extensions of Quill, including its applicability to other substructural type systems, such as affine or relevant typing (§6).

We begin with an introduction to linear type systems and their uses (§2), and conclude by discussing related work (§7).

## 2. Substructural Type Systems

Before describing the details of our language, we give examples of several applications of substructural type systems and several general-purpose substructural calculi.

### 2.1 Applications of Substructural Typing

Linear type systems restrict the use of weakening (i.e., discarding variables) and contraction (i.e., reusing variables), allowing us to reason about state and resource usage in programs. For example, excluding weakening could be used to prevent memory or resource leaks, by insisting that each input to a computation be consumed during its evaluation. Excluding contraction could be used in describing component layouts in circuits, where a limited number of each computational unit are available. Linear type systems combine these, providing exact control over which resources are used, while the exponential modality provides a mechanism to allow traditional (intuitionistic) notions of computation within a linear framework. This section describes two uses of linear types: session types and referentially-transparent in-place update. These demonstrate two different uses of linear types: session types evolve over the course of a computation, capturing changes in underlying state, while mutable values must be used linearly to preserve referential transparency. In each case, we will see both the need for the restrictions introduced by linearity, and the need to integrate linear and traditional modes of computation.

***Mutable arrays.*** We begin by considering in-place update. Suppose that we want to be able to read from and update arrays in a referentially transparent way. We might expect each update to produce a new copy of the array: otherwise, updates would be visible through other references to the original array. For large arrays, this copying will be extremely costly, both in time and space. The copying could be avoided if we could ensure that the use of arrays was single-threaded. That is, so long as no "old" copies of arrays are ever used, updates can be performed in place. Chen and Hudak [5] consider the connection between single-threaded usage, potentially enforced monadically, and linearity. They introduce an affine type system (they allow discarding but not duplication), and show that updating linearly-typed values can safely be performed in place. They also show how the operations on a linear data type can be interpreted to give a monad in an intuitionistic calculus, while preserving the safety of in-place update. However, this approach relies on hiding the linearly typed values, making them second-class citizens of the non-linear calculus. For example, their approach applies to linear arrays of non-linear element type, but not to linear arrays of linear element type.

***Session types.*** Next we consider session types, an instance of behavioral typing. Communication protocols frequently specify not just what data can be sent, but in what order. For example, the Simple Mail Transfer Protocol specifies not just a list of commands (identifying senders, recipients, message bodies, and so forth), but also a particular ordering to messages (the sender's address must precede the recipients' addresses, which must precede the message body). Session types, originally proposed by Honda [13], provide a mechanism for capturing such expectations in the types of communication channels. The critical aspect of his type system is that types evolve over the course of a computation to reflect the communication that has already taken place. For example, if channel $c$ has session type $\mathit{Int} \,!\, \mathit{Int} \,?\, \mathsf{End}$, we expect to send an integer along $c$, then receive an integer from $c$. After we have sent an integer, the type of $c$ must change to $\mathit{Int}?\mathsf{End}$, reflecting the remaining expected behavior. We can implement session types in a functional setting by giving channels linear types, and reflecting the evolution of types in the type signatures of the communication primitives:

$$\mathsf{send} :: t \otimes (t \,!\, s) \to s \qquad \mathsf{receive} :: (t \,?\, s) \to t \otimes s$$

Continuing the example above, we see that the result of $\mathsf{send}\ (4, c)$ will be of type $\mathit{Int} \,?\, \mathsf{End}$, as we hoped. The linearity of these channels is crucial to assuring the type correctness of communication: reusing channel $c$ would allow us to send arbitrarily many integers, not just one. There are approaches to encoding session types in existing functional languages, such as that of Pucella and Tov [23], but they result in channels being second class values. For example, sending or receiving channel names requires different primitives from those for sending or receiving other values.

### 2.2 General-Purpose Linear Calculi

Wadler [29] gives a $\lambda$-calculus based on Girard's logic of unity, a refinement of linear logic. In his approach, the types (ranged over by $\tau, \upsilon$) are precisely the propositions of linear logic, including pairs ($\tau \otimes \upsilon$), functions ($\tau \multimap \upsilon$), and the exponential modality ($!\tau$). His type system tracks two kinds of assumptions, linear $x : \langle \tau \rangle$ and intuitionistic $x : [\tau]$; only the latter are subject to contraction and weakening. Wadler does not include polymorphism in his calculus; nevertheless, we can see that his treatment of intuitionistic types would preclude attempts toward generality. He gives explicit term constructors to introduce and eliminate the exponential modality, and these constructs surround any use of unrestricted types. If $M$

is of type $\tau \multimap \upsilon$, and $N$ is of type $\tau$, then we can construct the application $M\,N$ of type $\upsilon$; on the other hand, if $M$ is of type $!(\tau \multimap \upsilon)$, then we must explicitly eliminate the ! constructor at each use of $M$, as $\underline{\texttt{let}}\ !f = M\ \underline{\texttt{in}}\ f\,N$. Returning to our introductory example, we have two families of types (and corresponding terms) for the $K$ combinator, $!(!\tau \multimap !(!\upsilon \multimap !\tau))$ if the first argument is intuitionistic, and $!(\tau \multimap !\upsilon \multimap \tau)$ otherwise.

Mazurak et al. [20] present a streamlined, polymorphic linear calculus. Their calculus, called F$^\circ$, extends the Girard-Reynolds polymorphic $\lambda$-calculus with linearity, and introduces a kind system which distinguishes between linear (kind $\circ$) and unrestricted (kind $\star$) types. They then define the kinds of types such as pairs in terms of the kinds of their components: $\tau \otimes \upsilon$ is of kind $\star$ if both $\tau$ and $\upsilon$ are of kind $\star$, and must be of kind $\circ$ otherwise. Finally, they introduce a subkinding relation, allowing a type of kind $\star$ to be used any place a type of kind $\circ$ is expected. This reflects the observation that a unrestricted value can be used any number of times, including once. While their approach seamlessly encompasses many uses of unrestricted types, it does not extend to functions. F$^\circ$ distinguishes between linear functions $\lambda^\circ x.M$, of type $\tau \xrightarrow{\circ} \upsilon$, which may capture arbitrary variables in their environment, and unrestricted functions $\lambda^\star x.M$, of type $\tau \xrightarrow{\bullet} \upsilon$, which can only capture unrestricted values. Consequently, F$^\circ$ still has four distinct types for the $K$ combinator

$$\forall(t:\circ).\forall(u:\star).t \xrightarrow{\bullet} u \xrightarrow{\circ} t \qquad \forall(t:\circ).\forall(u:\star).t \xrightarrow{\circ} u \xrightarrow{\circ} t$$
$$\forall(t:\star).\forall(u:\star).t \xrightarrow{\bullet} u \xrightarrow{\bullet} t \qquad \forall(t:\star).\forall(u:\star).t \xrightarrow{\circ} u \xrightarrow{\bullet} t$$

each with distinct inhabitants. The problem is endemic to the use of higher-order functions; for example, their system has numerous distinct application and composition functions.

Tov and Pucella [27] present Alms, an affine calculus with a kind system similar to F$^\circ$ but with additional flexibility in the treatment of functions. Their treatment of functions includes not just affine ($\xrightarrow{\mathsf{A}}$) and unrestricted ($\xrightarrow{\mathsf{U}}$) functions, but also functions with relative qualifiers. For example, Alms has a single most-general type for the $K$ combinator, written

$$\forall(t:\mathsf{A}).\forall(u:\mathsf{U}).t \xrightarrow{\mathsf{U}} u \xrightarrow{t} t.$$

The arrow $\xrightarrow{t}$ must be more restricted than the instantiation of $t$. If $t$ is instantiated to an affine type, then $\xrightarrow{t}$ must be $\xrightarrow{\mathsf{A}}$; otherwise, it can be $\xrightarrow{\mathsf{U}}$. They include subtyping explicitly; for example, $\mathit{Int} \xrightarrow{\mathsf{U}} \mathit{Int}$ is a subtype of $\mathit{Int} \xrightarrow{\mathsf{A}} \mathit{Int}$. Despite the (not insignificant) complexity of their system, it is still not clear that it fully supports the expressiveness of traditional functional programming languages. For example, their system has distinct composition operators with the types:

$$\forall(t:\mathsf{A})(u:\mathsf{A})(v:\mathsf{A}).(t \xrightarrow{\mathsf{A}} u) \xrightarrow{\mathsf{U}} (u \xrightarrow{\mathsf{A}} v) \xrightarrow{\mathsf{A}} (t \xrightarrow{\mathsf{A}} v)$$
$$\forall(t:\mathsf{A})(u:\mathsf{A})(v:\mathsf{A}).(t \xrightarrow{\mathsf{U}} u) \xrightarrow{\mathsf{U}} (u \xrightarrow{\mathsf{A}} v) \xrightarrow{\mathsf{U}} (t \xrightarrow{\mathsf{A}} v)$$

These types are not related by the subtyping relation, as subtyping is contravariant in function arguments.

## 3. Programming in Quill

This section gives an intuitive overview of our calculus Quill and its primary features. We begin by describing the use of overloading to capture the non-linear use of assumptions. We then consider the particular problems arising from having both linear and unrestricted functions, the overloading of application and abstraction, and introduce the corresponding predicates on types. Finally, we consider two examples of programming in Quill: a simple presentation of dyadic session types, demonstrating the use of linearity, and a Haskell-like presentation of monads, demonstrating the interaction between linearity and higher-order functional programming.

For the purposes of this section, we use a Haskell-like syntax for Quill, in which we distinguish linear functions ($\tau \xrightarrow{\circ} \upsilon$) from unrestricted functions ($\tau \xrightarrow{\bullet} \upsilon$). We give a formal account of Quill's syntax and semantics in the following sections.

### 3.1 Contraction and Weakening with Class

Our goal is a functional language in which values of some (but not all) types must be treated linearly. The central problem is the integration of unrestricted types, and functions on unrestricted (but otherwise generic) types, with an otherwise linear type system. We describe one solution, based on the theory of qualified types.

We begin by distinguishing linear from unrestricted types. We consider a type to be unrestricted if values of that type can be duplicated and discarded. That is, a type $\tau$ is unrestricted if we can exhibit values of type $\tau \xrightarrow{\bullet} 1$ and $\tau \xrightarrow{\bullet} \tau \otimes \tau$. (This approach roughly parallels Filinski's interpretation of intuitionistic types by commutative comonoids in the model of a linear calculus [7].) For example, consider a type for Booleans (perhaps implemented as $1 \oplus 1$) with the standard branching construct and constants. We can demonstrate that Booleans are unrestricted by giving the terms $\lambda b.\underline{\texttt{if}}\ b\ \underline{\texttt{then}}\ ()\ \underline{\texttt{else}}\ ()$ to discard a Boolean, and $\lambda b.\underline{\texttt{if}}\ b\ \underline{\texttt{then}}\ (\mathit{True},\mathit{True})\ \underline{\texttt{else}}\ (\mathit{False},\mathit{False})$ to copy one. This leaves the problem of how to write code generic over such types; for instance, we would like the function $\lambda x.(x, x+1)$ to be applicable to arguments of any unrestricted numeric type.

Our approach is inspired by the use of type classes in Haskell. Type classes were introduced to solve similar problems, such as how to write functions generic over types that have an equality operator, or that can be converted to and from text. For our purposes, we can imagine introducing a type class `Un`, which identifies unrestricted types:

```
class Un t where
  drop :: t ->• 1
  dup  :: t ->• t ⊗ t
```

The methods of `Un` provide the defining behavior of an unrestricted type. We could then imagine using these methods to implement terms such as the one above, for which we could write $\lambda x.\underline{\texttt{let}}\ (x,x') = \texttt{dup}\,x\ \underline{\texttt{in}}\ (x, x'+1)$. In inferring a type for this term, we would observe that its argument type has to support numeric operations (and so be a member of the `Num`) class, and has to support `dup` (and so be a member of the `Un` class). We would conclude that it should have type $(\texttt{Num}\,t, \texttt{Un}\,t) \Rightarrow t \xrightarrow{\bullet} t \otimes t$.

One advantage of this view of unrestricted types is that it extends naturally to products, sums, and recursive types. For example, a pair of values $(V, W)$ can safely be copied only when both $V$ and $W$ could individually be copied. We can capture this in an instance of the `Un` class:

```
instance (Un t, Un u) ⇒ Un (t ⊗ u) where
  drop (x, y)  =  let () = drop x in drop y
  dup (x, y)   =  ((x', y'), (x'', y'')) where
    (x', x'')  =  dup x
    (y', y'')  =  dup y
```

The relationship between the linearity of $t$ and $u$ and the linearity of $t \otimes u$ arises organically from the typing of the `drop` and `dup` methods. The argument for sums is parallel, with the same results.

Of course, we do not intend programmers to use the `drop` and `dup` methods directly, and we imagine that instances of `Un` would be inferred automatically from type declarations. Instead, Quill allows variables to be used freely, and infers `Un` predicates as if any duplication or discarding of variables had been done explicitly. Thus, $\lambda x.(x, x+1)$ is a well-typed Quill term with the type $(\texttt{Num}\,t, \texttt{Un}\,t) \Rightarrow t \xrightarrow{\bullet} t \otimes t$, as above.

### 3.2 The Problem of the Copyable Closure

We have an appealing view of how to distinguish unrestricted from linear types, and how to account for the linearity of products and sums. Unfortunately, this view does not extend to provide a uniform treatment of functions. Consider the curried pair constructor $\lambda x.\lambda y.(x,y)$. We know that the linearity of the resulting pair depends only on the linearity of its components. But what about the intermediate result? Suppose that we apply this function to some value $V$ of type $\tau$, giving the term $\lambda y.(V,y)$. Whether we can copy this term depends upon the captured value $V$; intuitively, we can say it depends on the function's closure. However, this is not reflected in the function type. (While $\tau$ does appear in the result type, so does $\upsilon$, but the linearity of the function type is solely a consequence of $\tau$.) We are thus forced to introduce distinct types for linear and unrestricted functions. This section discusses the resulting language design questions: how to handle application and abstraction in a language with multiple function types, and how to relate the type of a function to the type of its captured environment.

#### Application

We begin with application, the simpler of the two problems. Consider the uncurried application function. In intuitionistic calculi, this is $\lambda(f,x).f\,x$, of type $(a \to b, a) \to b$. In the linear settings, things are not so simple: we must decide whether the argument $f$ and the function being defined are linear or unrestricted functions. These choices are independent, giving four incomparable types:

$$(t \xrightarrow{\circ} u) \otimes t \xrightarrow{\circ} u \qquad (t \xrightarrow{\circ} u) \otimes t \xrightarrow{\bullet} u$$
$$(t \xrightarrow{\bullet} u) \otimes t \xrightarrow{\circ} u \qquad (t \xrightarrow{\bullet} u) \otimes t \xrightarrow{\bullet} u$$

We can resolve this by repetition by observing that (built-in) application is implicitly overloaded: we would like to write $f\,x$ whether $f$ is a linear or unrestricted function. We can make this overloading explicit in the types. We introduce a new predicate, $\mathsf{Fun}\,f$, which holds when $f$ is a function type; intuitively, we can think of this as corresponding to a class whose sole method is application, and whose only members are $\xrightarrow{\circ}$ and $\xrightarrow{\bullet}$. We can then type application with reference to this class, rather than in terms of either of the concrete function types. This reduces the number of application functions from four to two: we have $\mathsf{Fun}\,f \Rightarrow f\,t\,u \otimes t \xrightarrow{\circ} u$ and $\mathsf{Fun}\,f \Rightarrow f\,t\,u \otimes t \xrightarrow{\bullet} u$. We introduce syntactic sugar to make the $\mathsf{Fun}$ predicate easier to read. We will write $t \xrightarrow{f} u$ to indicate the type $f\,t\,u$ constrained by $\mathsf{Fun}\,f$, and further write $t \to u$ to indicate $t \xrightarrow{f} u$ for some fresh type variable $f$. Using this sugar, we arrive at the most general type for the application function, $(t \to u) \otimes t \to u$.

In the previous section, we motivated the typing of contraction and weakening using the methods of an `Un` class, even though we intend their use to be implicit in practice. In the case of the `Fun` predicate, the class method intuition is less helpful. Defining primitive application as a class method is difficult (how would it be used, except by application?), and we will rely on the `Fun` predicate holding only for the built-in function types. This reinforces the expressiveness of qualified types, even beyond their traditional application to overloaded class methods.

#### Abstraction

We have accounted for the uncurried application function. Now consider its curried equivalent, expressed intuitionistically as $\lambda f.\lambda x.f\,x$ of type $(t \to u) \to t \to u$. The problem here is similar to the problem with the $K$ combinator or the curried pair constructor. Suppose that we apply this function to some value $V$, giving $\lambda x.V\,x$: whether this function needs to be linear depends on the linearity of $V$. We thus have six incomparable types for the curried application function:

$$(t \xrightarrow{\circ} u) \xrightarrow{\circ} t \xrightarrow{\circ} u \qquad (t \xrightarrow{\bullet} u) \xrightarrow{\circ} t \xrightarrow{\circ} u \qquad (t \xrightarrow{\bullet} u) \xrightarrow{\circ} u \xrightarrow{\bullet} u$$
$$(t \xrightarrow{\circ} u) \xrightarrow{\bullet} t \xrightarrow{\circ} u \qquad (t \xrightarrow{\bullet} u) \xrightarrow{\bullet} t \xrightarrow{\circ} u \qquad (t \xrightarrow{\bullet} u) \xrightarrow{\bullet} u \xrightarrow{\bullet} u$$

Our approach to overloading application allows us to give names to individual function arrows in a type. Unfortunately, even this is not sufficient to account for the types of the application function; it only allows us to reduce the six types above to two:

$$(t \xrightarrow{f} u) \to t \xrightarrow{f} u \qquad (t \xrightarrow{\bullet} u) \to t \xrightarrow{\circ} u$$

However, this observation suggests our actual solution. Consider a more general type, subsuming the two above (but admitting one erroneous case): $(t \xrightarrow{f} u) \to t \xrightarrow{g} u$. The first case above is where $f$ and $g$ are the same type, and the second is where $f$ is less restricted (i.e., admits more structural rules) than $g$. We introduce a new predicate, $\tau \geq \upsilon$, which holds when $\tau$ admits more structural rules than $\upsilon$. We can now give the principal type of the application operator: $f \geq g \Rightarrow (t \xrightarrow{f} u) \to t \xrightarrow{g} u$.

Our examples have focused on function types. However, the $\geq$ relation is not limited to functions; for example, consider the possible types of the curried pair constructor $\lambda x.\lambda y.(x,y)$:

$$\mathsf{Un}\,t \Rightarrow t \to u \to t \otimes u \qquad t \to u \xrightarrow{\circ} t \otimes u$$

As for the application operator, we see that the linearity of the final arrow is restricted by the types appearing before it in the type signature. Unlike in that case, however, the earlier type in question is not a function type. We can give the curried pair constructor the principal type $t \geq f \Rightarrow t \to u \xrightarrow{f} t \otimes u$.

### 3.3 Quill in Action

One of the surprising aspects of this work has been the simplicity of our motivating examples: the $K$ combinator and application functions are very short, but reveal the unique benefits of Quill. We conclude this section by turning to several larger examples. First, we consider a simple embedding of dyadic session types, a typical application of linear typing. Doing so demonstrates that we have not made our system too permissive. Second, we consider a presentation of Haskell's monad class and several of its instances. This shows that Quill supports the full generality of intuitionistic functional programming abstractions, and demonstrates the additional information captured by a linear type system.

For these examples, we will assume various language features present in Haskell, such as new type definitions, multi-parameter type classes with functional dependencies, and `do` notation for monads. We believe these are representative of realistic settings for linear functional programming. However, these ideas are not fundamental to our approach, and our formalization in the following sections will consider a core calculus that does not assume such language features or syntactic sugar.

#### Dyadic Session Types

Session types [13] provide a typing discipline for communication protocols among asynchronous processes. There is a significant body of work exploring the combination of session-typed and functional programming. Much of this work has focused on defining new linear calculi, combining functional and concurrent programming [9, 19, 28]. These calculi frequently include details specific to session typing in their type systems, and so seem a poor fit for general purpose programming languages. Pucella and Tov [23] give an encoding of session types in Haskell, wrapping an underlying untyped use of channels. They express the session typing discipline using the existing features of the Haskell class system. However, they rely on providing only second-class channels as values, cap-

turing the session types of channels in a parameterized monad. One consequence of this is that sending and receiving channels, while possible, requires primitive operations (with particularly involved types) distinct from those for sending and receiving values. We will show that Quill allows us to have the best of both worlds: because Quill is linear, we can have first-class channels, and because Quill fits into the existing work on qualified types we can encode the session typing discipline without having to extend our core type system.

Honda gives five constructors for session types $\zeta$, interpreted as follows:

| | |
|---|---|
| $\tau\,!\,\zeta$ | Send a value of type $\tau$, then continue as $\zeta$ |
| $\tau\,?\,\zeta$ | Receive a value of type $\tau$, then continue as $\zeta$ |
| $\zeta \uplus \zeta'$ | Choose between behaviors $\zeta$ and $\zeta'$ |
| $\zeta \pitchfork \zeta'$ | Offer a choice of behaviors $\zeta$ and $\zeta'$ |
| `End` | No communication |

(Our syntax for the choice constructors differs from Honda's to avoid conflict with the notation for the linear logic connectives.) Lindley and Morris [19] observed that, in a linear functional setting, the choice types can be encoded in terms of $\oplus$ and the input and output types, and so we omit them from our example. We introduce types for the remaining session types—these types are empty, as we will use them as tags rather than to type channels directly.

```
data t :!: s
data t :?: s
data End
```

Honda observed that communicating processes had dual expectations for their shared channels: if one process expects to send a value of type $\tau$, the other process should expect to receive a value of type $\tau$. Following Pucella and Tov [23], we can capture this using a type class with functional dependencies [17]:

```
class Dual t u | t → u, u → t
instance Dual s s' ⇒ Dual (t :!: s) (t :?: s')
instance Dual s s' ⇒ Dual (t :?: s) (t :!: s')
instance Dual End End
```

We now turn to channels and their primitive operators.

```
data Ch s
instance Un (Ch End)
```

Unlike other approaches to encoding session types in functional languages, we treat `End` channels as unrestricted, avoiding the need for explicit `close` operations. Previous work on linearity has discussed the encapsulation of unrestricted types in linear ones, either via existential types [20, 27] or via a module system [23]. Alternatively, one might prefer to take the notion of linear channels as primitive. Either approach is possible in Quill; as we are primarily concerned with the use of linear types, we omit further discussion of them here. (But see the extended version of this paper for the details of the packaging approach.) The primitive operations on session-typed channels are as follows:

```
fork :: Dual s s' ⇒ (Ch s → M ()) → M (Ch s')
send :: t ≥ f ⇒ t → Ch (t :!: s) -f-> M (Ch s)
receive :: Ch (t :?: s) → M (t ⊗ Ch s)
```

We adopt the `fork` construct of Lindley and Morris for its simplicity and because it assures deadlock freedom. The `Dual` predicate assures that the session types `s` and `s'` are well-formed and dual. Gay and Vasconcelos [9] give two typings for the `send` function, depending on the linearity of its first argument:

$$t \xrightarrow{\bullet} (t\,!\,s) \xrightarrow{\bullet} s \quad \text{if } t \text{ is unrestricted}$$
$$t \xrightarrow{\bullet} (t\,!\,s) \xrightarrow{\circ} s \quad \text{otherwise}$$

This fits precisely the pattern captured by the $\geq$ predicate in Quill. Finally, as the communication primitives are side-effecting, we assume the results are embedded in some suitable monad `M`. This is not an entirely innocuous choice; we will return to monads in a linear setting for our next example.

We present a simple example using session-typed channels. We begin with a process that performs an arithmetic operation:

```
multiplier c =
  do (x, c) ← receive c
     (y, c) ← receive c
     send (x * y) c
     return ()
```

The `multiplier` function defines a process that expects to read two numbers on channel `c`, and then sends their product back along the same channel. The inferred type for `multiplier` is `Num t ⇒ Ch (t :?: (t :?: (t :!: End))) → M ()`. Note that, despite our reuse of the name `c`, each call to a communication primitive returns a new copy of the channel, which is used linearly. Next, we define a process to communicate with `multiplier`. To illustrate the use of channels as first-class values, we define it in a round-about way. First, we define a process that provides only one of the two expected values:

```
sixSender c =
  do (d, c) ← receive c
     send 6 d
     send d c
     return ()
```

This function defines a process that begins by receiving a channel `d` along `c`; it then sends 6 along the received channel before returning the received channel along `c`. Thus, its type is

```
Num t ⇒ Ch (Ch (t :!: s) :?: Ch s :!: End) → M ()
```

Finally, we can define the main process, which uses the preceding processes to compute 42:

```
answer = do d ← fork sixSender
            c ← fork multiplier
            d ← send c d
            (c, d) ← receive d
            c ← send 7 c
            (x, c) ← receive c
            return x
```

This example demonstrates the advantages of Quill for linear programming. Unlike encoding-based approaches, we have simple types and uniform treatment of channels and other data. Unlike other concurrency-focused approaches, we have not built any aspects of session typing into our language or its type system.

### Monads

In the previous example, we assumed that we could express our communication primitives monadically, to account for their side effects. As they are fundamentally reliant on higher-order functions, it is worth examining the interaction between linearity and the monadic combinators. For a simple example, consider the desugaring of `answer`, which begins

$$\texttt{fork sixSender} \gg\!= \backslash d \to \texttt{fork multiplier} \gg\!= \backslash c \to M$$

where $M$ denotes the remainder of `answer`, and both $c$ and $d$ are free in $M$. As $d$ is of linear type, we see that $\lambda c.M$ must be a linear function. Does this mean that the result of $\gg\!=$ must also be linear? How does this play out for other monads, like the `Maybe` monad?

Of course, we could transport standard intuitionistic definitions of monads directly into Quill, treating all functions as unrestricted. Doing so would allow us to use monads for unrestricted values

without any new complexity. However, doing so would also rule out interesting examples, such as those with channels in the previous example. Here we take the opposite perspective, attempting to generalize standard notions of monads to include the linear cases. We will consider two canonical examples, failure and state.

First, we consider failure. We assume we have some type `Maybe` $t$ with constructors `Just` and `Nothing`; observe that `Maybe` $t$ is unrestricted precisely when $t$ is unrestricted. To demonstrate that `Maybe` is a monad, we give implementations of the `return` and $(\gg\!=)$ operators, as follows:

```
return = \x → Just x
(>>=)  = \m → \f → case m of
                     Nothing → Nothing
                     Just x → f x
```

The typing of `return` is uninteresting. On the other hand, consider the use of $f$ in the body of $(\gg\!=)$: if $m$ is `Nothing`, then $f$ is discarded, whereas if $m$ is `Just` $x$, then $f$ is used once. So, we see that $f$ must be unrestricted, and so we have the types:

$$\texttt{return} :: t \to \texttt{Maybe}\, t$$
$$(\gg\!=) \;\; :: t \geq f \Rightarrow \texttt{Maybe}\, t \to (t \xrightarrow{\bullet} \texttt{Maybe}\, u) \xrightarrow{f} \texttt{Maybe}\, u$$

The requirement that $f$ be unrestricted captures that the remainder of the computation may not occur, an important characteristic of the failure monad. For example, this suggests that the monad in the previous example cannot include exceptions. This aligns with our expectations: if a process fails, it cannot fulfill its outstanding session-type obligations.

Next, we consider the state monad. A state monad for state values of type `S` is typically implemented in Haskell by the type $\texttt{S} \to (t, \texttt{S})$. This introduces additional choice in the linear case: should we consider values of type $\texttt{S} \xrightarrow{\circ} t \otimes \texttt{S}$ or of type $\texttt{S} \xrightarrow{\bullet} t \otimes \texttt{S}$? What constraints would this choice impose on the use of the monad? We can clarify these questions by considering the definition of `return` and $(\gg\!=)$. (Relying on our generalization of abstraction and application, we consider these implementations in parallel with the choice of the state monad itself.)

```
return = \x → \s → (x, s)
(>>=)  = \m → \f → \s → let (x, s') = m s in f x s
```

We make two observations about the state monad. First, $x$ is captured in $\backslash s \to (x, s)$; therefore, a state computation can only be as unrestricted as its result values. (This is true of the failure monad as well, but is reflected in the inherent linearity of `Maybe` types.) Second, note that the function $f$ is used linearly in the body of $(\gg\!=)$, so its type need not be unrestricted (unlike for the failure monad). These observations are reflected in the types of `return` and $(\gg\!=)$. We begin by introducing an alias for the state monad type:

```
type State m s t = s →[m] (t ⊗ s)
```

We can then type `return` and $(\gg\!=)$ by

```
return :: t ≥ State m s ⇒ t → State m s t
(>>=)  :: (State m s ≥ g, f ≥ State m s) ⇒
          State m s t →[f] (t → State m s u) →[g]
          State m s u
```

The predicate `State` $m \geq g$ reflects that the term $\backslash f \to \ldots$ has captured $m$ of type `State` $m\, s\, t$.

Finally, we generalize these examples. The problem is the type of the second argument to $(\gg\!=)$: to be useful in the linear context, we must sometimes include the restricted function type, but to incorporate the full range of monads we must sometimes limit it to unrestricted functions. We encompass both cases using a multi-parameter type class for monads:

| | | | |
|---|---|---|---|
| Term variable | $x, y \in \mathit{Var}$ | Type variables | $t, u \in \mathit{TVar}$ |
| Multienvironments | $H$ | Environments | $\Gamma, \Delta$ |
| Type constructors | $T^\kappa \in \mathcal{T}^\kappa$ where $\{\oplus, \xrightarrow{\bullet}, \xrightarrow{\circ}\} \subseteq \mathcal{T}^{\star\to\star\to\star}$ | | |

$$\begin{array}{lrl}
\text{Kinds} & \kappa & ::= \star \mid \kappa \to \kappa \\
\text{Types} & \tau^\kappa & ::= t \mid T^\kappa \mid \tau^{\kappa'\to\kappa}\, \tau^{\kappa'} \\
\text{Predicates} & \pi & ::= \mathsf{Un}\, \tau \mid \mathsf{Fun}\, \tau \mid \tau \geq \upsilon \\
\text{Qualified types} & \rho & ::= \tau^\star \mid \pi \Rightarrow \rho \\
\text{Type schemes} & \sigma & ::= \rho \mid \forall t.\sigma \\
\text{Expressions} & M, N & ::= x \mid K\, M \mid \lambda x.M \mid M\, N \mid \mathsf{in}_1\, M \mid \mathsf{in}_2\, N \\
 & & \mid \underline{\mathsf{case}}\, M\, \underline{\mathsf{of}}\, \{\mathsf{in}_1\, x \mapsto N;\ \mathsf{in}_2\, y \mapsto N'\} \\
 & & \mid \underline{\mathsf{let}}\, x = M\, \underline{\mathsf{in}}\, N \mid \underline{\mathsf{let}}\, K\, x = M\, \underline{\mathsf{in}}\, N
\end{array}$$

**Figure 1:** Quill types and terms.

```
class Monad f m | m → f where
  return :: t ≥ m t ⇒ t → m t
  (>>=)  :: (m t ≥ g, f ≥ m g) ⇒
            m t → (t →[f] m u) →[g] m u
```

The definitions above give instances of our new `Monad` class:

```
instance Monad (→•) Maybe
instance Monad m (State m s)
```

and that the example of dyadic session types will type in monads $m$ such that `Monad` $(\xrightarrow{\circ})$ $m$ is provable.

## 4. Substructural Qualified Types

We have considered some of the challenges of using linear calculi in practice, given an intuitive description of our solution, based on qualified types, and demonstrated how that solution might be realized in a Haskell-like practical programming language. In this section, we give a formal account of our approach to substructural qualified types. We begin by giving an overview of a core Quill calculus and its type system (§4.1). We then give a syntax-directed variant on the type system (§4.2), preparatory to giving an Algorithm $\mathcal{M}$ style type inference algorithm (§4.3). Finally, we relate Quill typing to typing for a non-substructural core calculus (§4.4), making concrete our claims that Quill encompasses existing functional programming practice.

### 4.1 Quill Terms and Typing

The syntax of Quill types and terms is shown in Figure 1. Quill types are stratified according to a simple kind system; we write $\tau$, $\upsilon$ and $\phi$ (without superscripts) to range over types of any kind. (Unlike, we use kind $\star$ for all types, not just unrestricted ones.) We assume that $\xrightarrow{\circ}$, $\xrightarrow{\bullet}$ and $\oplus$ are binary type constructors, which we will write infix, corresponding to linear and unrestricted functions and additive sums. We do not include multiplicative or additive products ($\tau \otimes \upsilon$ and $\tau \,\&\, \upsilon$), as these can be encoded in terms of the other types. (These encodings depend on our overloading of abstraction for their full generality.) We allow arbitrary additional type constructors, providing other (user-defined) datatypes. Datatypes capture first-class universal and existential types, following the approach of Jones's FCP [14]. While we have not used these features in our examples, existential types are used prominently in other approaches to linear functional programming [20, 27], particularly to construct linear wrappers around unrestricted types, and so we show that our system can accommodate them as well. Predicates $\pi$ include those necessary for our treatment of linearity (and can constrain higher-kinded types). Qualified types and type schemes are standard for overloaded Hindley-Milner calculi. We write $\forall \vec{t}.Q \Rightarrow \tau$ to abbreviate $\forall t_1 \ldots \forall t_n.P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow \tau$.

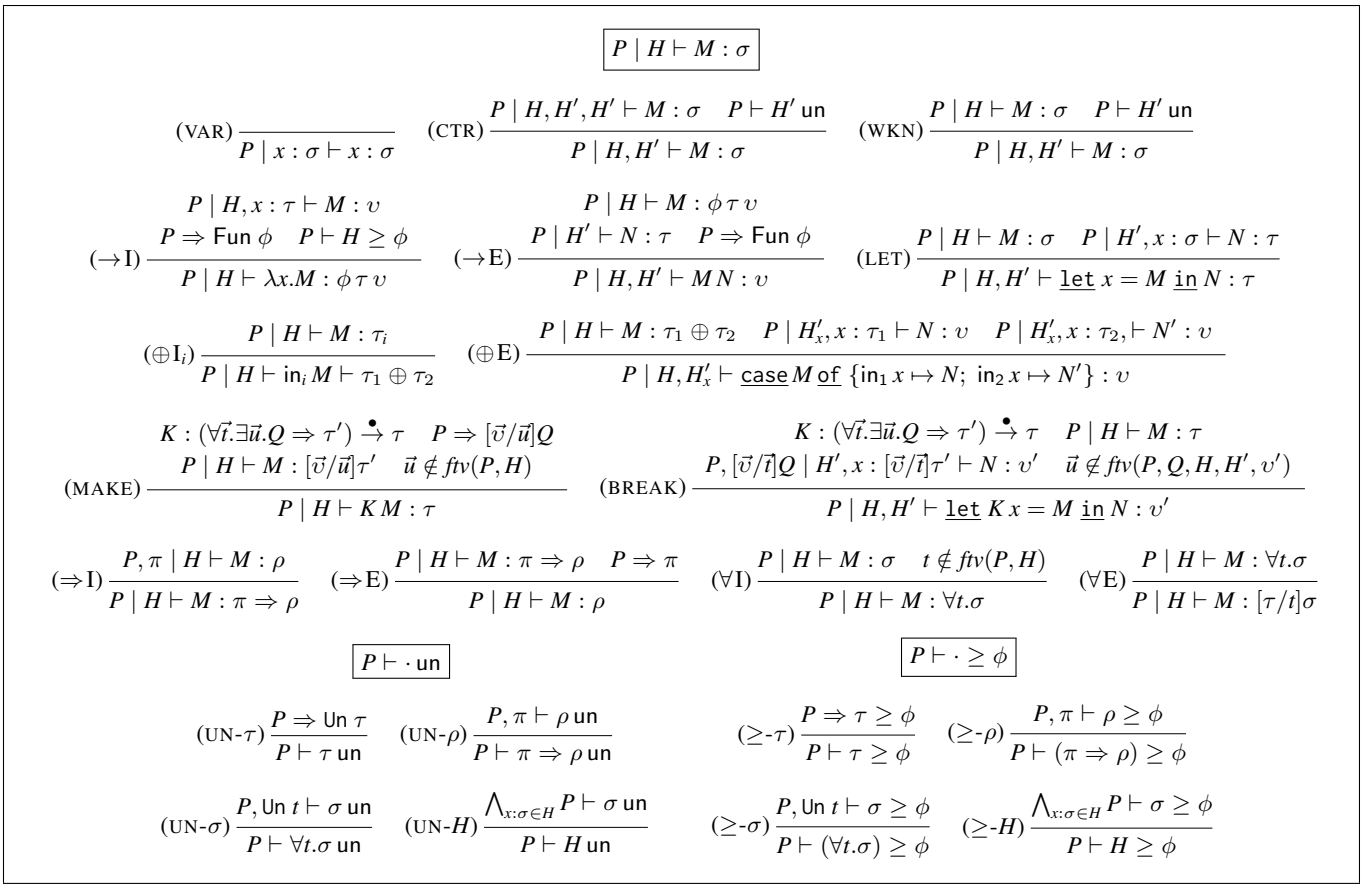


**Figure 2:** Typing rules.

Quill includes standard terms for variables, abstractions, applications, and the additive sums. We introduce polymorphism at $\underline{\mathtt{let}}$ bindings. First-class existential and universal types are expressed using constructors $K$. We assume an ambient signature mapping individual constructors $K$ to types $\forall v.(\forall \vec{t}.\exists \vec{u}.Q \Rightarrow \tau') \to \tau$. (This type is not included in $\sigma$; $\sigma$ denotes inferable type schemes.) Construction $K\,M$ builds a value of type $\tau$, assuming that $\tau'$ has a suitably generic type. We insist that constructors be fully applied. Deconstruction $\underline{\mathtt{let}}\ K\,x = M\ \underline{\mathtt{in}}\ N$ eliminates such values. This treatment of datatypes corresponds to common practice in Haskell, and allows us to make clear the extent of type inference.

Figure 2 gives the Quill type system. The typing judgment is $P \mid H \vdash M : \sigma$, where $P$ is a set of predicates on the type variables in the remainder of the judgment, $H$ is a typing environment, $M$ is a term and $\sigma$ a type scheme. The typing judgment is linear: in (VAR), there must only be one binding in the environment, while rules like ($\to$ E) split the typing environment among their hypotheses. To account for multiple uses of a single variable, we use multisets of typing assumptions (which we call "multienvironments"); thus, the multienvironment $\{\!|x : \sigma, x : \sigma|\!\}$ is distinct from $\{\!|x : \sigma|\!\}$. We do insist that multienvironments be consistent, so if $\{\!|x : \sigma, x : \sigma'|\!\} \subseteq H$ then we must have $\sigma = \sigma'$. Assumptions of unrestricted type are duplicated in (CTR) and discarded in (WKN). Both rules use the auxiliary judgment $P \vdash -\,\mathsf{un}$, which lifts the $\mathsf{Un}$ predicate (and the predicate entailment relation) to typing environments. The assumption $\mathsf{Un}\ t$ in (UN-$\sigma$) accounts for terms like the empty list, which should be treated as unrestricted until its type variables are instantiated. Rules ($\to$ I) and ($\to$ E) implement overloading of abstraction and application. In ($\to$ E), note that we allow the function term to be of any type $\phi$, so long as it satisfies the constraint $\mathsf{Fun}\ \phi$. In ($\to$ I), we allow a term $\lambda x.M$ to have any function type, so long as that type is more restricted than its environment; the auxiliary judgment $P \vdash - \geq \phi$ lifts the $\geq$ predicate to type environments. We will assume throughout this presentation that binders introduce fresh names. First-class polymorphism is introduced in (MAKE) and eliminated in (BREAK); our approach follows Jones [14] almost exactly, but adds a predicate context $Q$. We write $K : (\forall t.\exists u.Q \Rightarrow \tau') \to \tau$ to denote an instantiation of the signature for $K$ in the ambient context. We assume that each datatype has at most one constructor; more complex datatypes can be expressed using the other features of the type system. The remaining rules are standard for linear sums and qualified polymorphism.

Figure 3 gives a minimal definition of the predicate entailment relation $P \Rightarrow Q$. One strength of type systems based on qualified types is that the predicate system provides a natural point of extension, and our approach here is no different. Nevertheless, we specify some rules for the entailment judgment, namely the linearity of the built-in types and that (only) $\xrightarrow{\circ}$ and $\xrightarrow{\bullet}$ are in class $\mathsf{Fun}$. In determining the linearity of a datatype $\tau$, we assume that the universally quantified type variables $\vec{t}$ are unrestricted (as a term of type $\tau$ cannot have made any assumptions of $\vec{t}$), but cannot do so for the existentially quantified variables $\vec{u}$, as they may have been instantiated arbitrarily in constructing the $\tau$ value. Our definition of $\geq$ is sufficient for this presentation (such predicates are only introduced with function types on their right-hand sides), but less general than might be expected. We will return to the definition of this class when we discuss extensions to Quill (§6). Finally, the lifting

$$\frac{P \ni \pi}{P \Rightarrow \pi} \qquad \frac{\bigwedge_{\pi \in Q} P \Rightarrow \pi}{P \Rightarrow Q} \qquad \frac{\tau = \overset{\circ}{\to} \vee \tau = \overset{\bullet}{\to}}{P \Rightarrow \mathsf{Fun}\ \tau}$$

$$\frac{K : (\forall \vec{t}. \exists \vec{u}. Q \Rightarrow \tau') \overset{\bullet}{\to} \tau \qquad P, Q, \mathsf{Un}\ \vec{t} \Rightarrow \mathsf{Un}\ \tau'}{P \Rightarrow \mathsf{Un}\ \tau}$$

$$\frac{}{P \Rightarrow \mathsf{Un}\ (\tau \overset{\bullet}{\to} \upsilon)} \qquad \frac{P \Rightarrow \mathsf{Un}\ \tau_1 \qquad P \Rightarrow \mathsf{Un}\ \tau_2}{P \Rightarrow \mathsf{Un}\ (\tau_1 \oplus \tau_2)}$$

$$\frac{P \Rightarrow \mathsf{Un}\ \tau}{P \Rightarrow \tau \geq (\upsilon \overset{\bullet}{\to} \upsilon')} \qquad \frac{}{P \Rightarrow \tau \geq (\upsilon \overset{\circ}{\to} \upsilon')}$$

$$\frac{P \Rightarrow \tau \geq \phi\, t \quad t \text{ fresh}}{P \Rightarrow \tau \geq \phi} \qquad \frac{P \Rightarrow \tau\, t \geq \phi \quad t \text{ fresh}}{P \Rightarrow \tau \geq \phi}$$

**Figure 3:** Entailment rules.

cases for $\geq$ are a notational convenience; for example, they allows us to write $\tau \geq \phi$ rather than $\tau \geq \phi\, t\, u$ for fresh $t$ and $u$.

### 4.2 A Syntax-Directed Quill Type System

The Quill type system has a number of rules that are not syntax directed, including the structural rules and the rules introducing and eliminating polymorphism. To simplify the definition of type inference and the proofs of its correctness, we give a syntax-directed variant of the Quill type system. In doing so, we address two independent concerns. First, the rules ($\forall$ I), ($\forall$ E), ($\Rightarrow$ I), and ($\Rightarrow$ E) may be used at any point in a derivation. This problem has already been studied in the general context of qualified types. An identical solution applies in Quill: uses of ($\forall$ E) and ($\Rightarrow$ E) may always be permuted to occur at occurrences of (VAR), while uses of ($\forall$ I) and ($\Rightarrow$ I) may always be permuted to occur at occurrences of (LET) or at the end of the derivation. Second, the structural rules (WKN) and (CTR) may also appear at any point in a typing derivation. As in the polymorphism cases, we show that uses of these rules can be permuted to definite places in the derivation: uses of (CTR) can be permuted to appear immediately below a rule with multiple hypotheses (such as ($\to$ E) or ($\oplus$ E)) and uses of (WKN) can be permuted to occurrences of (VAR).

Figure 4 gives the syntax-directed variant of the Quill system. The judgment $P \mid \Gamma \vdash^{S} M : \tau$, is a syntax-directed variant of $P \mid H \vdash M : \sigma$, and uses standard type environments $\Gamma$ rather than multienvironments $H$. The auxiliary judgments are used unchanged. The syntax-directed system differs from the original type system in two ways. First, we account for polymorphism. Our approach is identical to Jones's approach for (intuitionistic) qualified types [15]: we introduce instantiation and generalization operators, accounting for the role of predicates, and collapse the treatment of polymorphism into the instances of (VAR$^{S}$) and (LET$^{S}$).

**Definition 1.** We define instantiation and generalization as follows, where we write $[\tau_i/t_i]$ for the substitution $[\tau_1/t_1, \ldots, \tau_n/t_n]$.

1. Let $\sigma$ be some type scheme $\forall \vec{t}. P \Rightarrow \tau'$. We say that $Q \Rightarrow \tau$ is an instance of $\sigma$, written $(Q \Rightarrow \tau) \sqsubseteq \sigma$, if there is some $\vec{\upsilon}$ such that $\tau = [\upsilon_i/t_i]\tau'$ and $Q \Rightarrow [\upsilon_i/t_i]P$.
2. Let $\Gamma$ be a typing environment, and $\rho$ a qualified type. We define $Gen(\Gamma, \rho)$ to be the type scheme $\forall (ftv(\rho) \setminus ftv(\Gamma)).\rho$.

We use instantiation in (VAR$^{S}$), collapsing a use of (VAR) and subsequent uses of ($\forall$ E) and ($\Rightarrow$ E), and generalization in (LET$^{S}$) collapsing a use of (LET) and preceding uses of ($\Rightarrow$ I) and ($\forall$ I). Second, we account for contraction and weakening. In (VAR$^{S}$), we allow an arbitrary environment, so long as the unused assumptions $\Delta$ are unrestricted. In ($\to$E$^{S}$), we partition the input environment into three parts: $\Gamma$ is used exclusively in typing $M$, $\Gamma'$ is used exclusively in typing $N$, and $\Delta$ is used in both; consequently, assumptions in $\Delta$ must be unrestricted. The remaining rules follow the same pattern.

The goal of the syntax-directed type system is a one-to-one correspondence between syntactic forms and typing rules. However, it is not the case that a typeable term has exactly one syntax-directed typing derivation. For example, while the contents of the linear environments are determined by the term structure, the contents of the unrestricted environments are not. For another example, consider the term $(\lambda x.x)\, y$. We can choose to type the abstraction as either $t \overset{\circ}{\to} t$ or $t \overset{\bullet}{\to} t$ (or even as $t \overset{f}{\to} t$ assuming the predicate $\mathsf{Fun}\, f$). Each of these choices would make the term well-typed, and we assume they introduce no observable semantic distinctions.

We now relate our original and syntax-directed type systems. We start with environments. Intuitively, the syntax-driven system introduces contraction whenever needed; however, a multienvironment $H$ could contain multiple instances of assumptions with linear types. We introduce a notion of approximation between multienvironments $H$ and environments $\Gamma$ that holds when the only repeated assumptions in $H$ are for unrestricted types.

**Definition 2.** If $H$ is a multienvironment, $\Gamma$ is an environment, and $P$ is some context, then we say that $\Gamma$ approximates $H$ under $P$, written $P \vdash H \approx \Gamma$, if:

- $x : \sigma \in H$ if and only if $x : \sigma \in \Gamma$; and,
- If $\{\!| x : \sigma, x : \sigma |\!\} \subseteq H$, then $P \vdash \sigma\ \mathsf{un}$.

We now turn to our primary results. First, derivations in the syntax-directed system correspond to derivations in the original system.

**Theorem 3** (Soundness of $\vdash^{S}$)**.** *If $P \mid \Gamma \vdash^{S} M : \tau$ and $P \vdash H \approx \Gamma$, then $P \mid H \vdash M : \tau$.*

The proof is by structural induction on the derivation of $P \mid \Gamma \vdash^{S} M : \tau$, and relies on introducing instances of the structural and polymorphism rules.

Second, we show completeness of the syntax directed system. A derivation in the original system may end with uses of ($\Rightarrow$ I) or ($\forall$I), moving predicates from the context to the type or quantifying over free type variables. In contrast, there are no such steps in a derivations in the syntax-directed system. To account for this difference, we introduce a notion of qualified type schemes, again following Jones [15].

**Definition 4.** A qualified type scheme $(P \mid \sigma)$ pairs a type scheme $\sigma$ with a set of predicates $P$. Let $\sigma$ be $\forall \vec{t}. P \Rightarrow \tau$ and $\sigma'$ be $\forall \vec{t'}. P' \Rightarrow \tau'$. We say that $(P \mid \sigma)$ is an instance of $(P' \mid \sigma')$, written $(P \mid \sigma) \sqsubseteq (P' \mid \sigma')$ iff there are $\vec{\upsilon}$ such that $\tau = [\upsilon_i/t'_i]\tau'$ and $P, Q \Rightarrow P', [\upsilon_i/t'_i]Q'$. We treat type schemes $\sigma$ as abbreviations for qualified type schemes $(\emptyset \mid \sigma)$.

We can now state the completeness of the syntax-directed system.

**Theorem 5** (Completeness of $\vdash^{S}$)**.** *If $P \mid H \vdash M : \sigma$ and $P \vdash H \approx \Gamma$, then there are some $Q$ and $\tau$ such that $Q \mid \Gamma \vdash^{S} M : \tau$ and $(P \mid \sigma) \sqsubseteq Gen(\Gamma, Q \Rightarrow \tau)$.*

Intuitively, this states that for any derivation in our original type system, there is a derivation of a more general result in the syntax-directed system. The proof is by induction on the derivation of $P \mid H \vdash M : \sigma$. The interesting cases rely on the role of generalization and instantiation in the syntax-directed type system and the safe movement of structural rules up derivation trees.

$$(\text{VAR}^{\text{S}})\ \frac{P \vdash \Delta\ \mathsf{un} \quad (P \Rightarrow \tau) \sqsubseteq \sigma}{P \mid \Delta, x : \sigma \vdash^{\text{S}} x : \tau} \qquad (\text{LET}^{\text{S}})\ \frac{Q \mid \Gamma, \Delta \vdash^{\text{S}} M : \tau \quad \sigma = \mathit{Gen}(\Gamma, \Delta; Q \Rightarrow \tau) \quad P \mid \Gamma', \Delta, x : \sigma \vdash^{\text{S}} N : \upsilon \quad P \vdash \Delta\ \mathsf{un}}{P \mid \Gamma, \Gamma', \Delta \vdash^{\text{S}} \underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N : \upsilon}$$

$$(\to \text{I}^{\text{S}})\ \frac{P \mid \Gamma, x : \tau \vdash^{\text{S}} M : \upsilon \quad P \Rightarrow \mathsf{Fun}\ \phi \quad P \vdash \Gamma \geq \phi \quad P \vdash \Delta\ \mathsf{un}}{P \mid \Gamma, \Delta \vdash^{\text{S}} \lambda x.M : \phi\, \tau\, \upsilon} \qquad (\to \text{E}^{\text{S}})\ \frac{P \mid \Gamma, \Delta \vdash^{\text{S}} M : \phi\, \tau\, \upsilon \quad P \mid \Gamma', \Delta \vdash^{\text{S}} N : \tau \quad P \Rightarrow \mathsf{Fun}\ \phi \quad P \vdash \Delta\ \mathsf{un}}{P \mid \Gamma, \Gamma', \Delta \vdash^{\text{S}} M\, N : \upsilon}$$

$$(\oplus \text{I}_i{}^{\text{S}})\ \frac{P \mid \Gamma \vdash^{\text{S}} M : \tau_i}{P \mid \Gamma \vdash^{\text{S}} \mathsf{in}_i\, M : \tau_1 \oplus \tau_2} \qquad (\oplus \text{E}^{\text{S}})\ \frac{P \mid \Gamma, \Delta \vdash^{\text{S}} M : \tau_1 \oplus \tau_2 \quad P \vdash \Delta\ \mathsf{un} \quad P \mid \Gamma', \Delta, x : \tau_1 \vdash^{\text{S}} N : \upsilon \quad P \mid \Gamma', \Delta, x : \tau_2 \vdash^{\text{S}} N' : \upsilon}{P \mid \Gamma, \Gamma', \Delta \vdash^{\text{S}} \underline{\mathsf{case}}\, M\, \underline{\mathsf{of}}\ \{\mathsf{in}_1\, x \mapsto N;\ \mathsf{in}_2\, x \mapsto N'\} : \upsilon}$$

$$(\text{MAKE})\ \frac{K : (\forall \vec{t}.\exists \vec{u}.Q \Rightarrow \tau') \overset{\bullet}{\to} \tau \quad P \Rightarrow [\vec{\upsilon}/\vec{u}]Q \quad P \mid \Gamma \vdash M : [\vec{\upsilon}/\vec{u}]\tau' \quad \vec{t} \notin \mathit{ftv}(P, H)}{P \mid \Gamma \vdash K\, M : \tau} \qquad (\text{BREAK})\ \frac{K : (\forall \vec{t}.\exists \vec{u}.Q \Rightarrow \tau') \overset{\bullet}{\to} \tau \quad P \mid \Gamma, \Delta \vdash M : \tau \quad P \vdash \Delta\ \mathsf{un} \quad P, [\vec{\upsilon}/\vec{t}]Q \mid \Gamma', \Delta, x : [\vec{\upsilon}/\vec{t}]\tau' \vdash N : \upsilon' \quad \vec{u} \notin \mathit{ftv}(P, Q, \Gamma, \Gamma', \Delta, \upsilon')}{P \mid \Gamma, \Gamma', \Delta \vdash \underline{\mathsf{let}}\ K\, x = M\ \underline{\mathsf{in}}\ N : \upsilon'}$$

**Figure 4:** Syntax-directed typing rules.

### 4.3 Type Inference for Quill

Having defined a suitable target type system, we can give a type inference algorithm for Quill. We have three separate concerns during type inference. First, we use a standard Hindley-Milner treatment of polymorphism. Second, we introduce `Un` predicates for non-linear use of variables. We track the variables used in each expression, and so identify reuse or discarding of variables. Third, we account for first-class polymorphism. We introduce a distinction between rigid and flexible type variables; only the latter are bound in unification. These three concerns add apparent complexity to the type inference algorithm, but can be understood separately.

The inference algorithm is given in Figure 5, in the style of Algorithm $\mathcal{M}$ [18]. The inputs include the environment $\Gamma$, expression $M$ and expected type $\tau$, along with the current substitution $S$ and the rigid type variables $X$. The output includes the generated predicates $P$, the resulting substitution $S'$, and a set of used (term) variables $\Sigma$. We let $u_i$ range over fresh type variables, and let $U, R, S$ range over substitutions. We will look at several illustrative cases of the algorithm in detail; the remaining cases are constructed along the same lines.

In the variable case, we are given both the variable $x$ and its expected type $\tau$. We unify $x$'s actual type, given by $\Gamma$, with its expected type $\tau$. This illustrates the primary difference between this approach and Milner's type inference algorithm $\mathcal{W}$: we move unification as close to the leaves as possible. We defer the details of unification to a separate algorithm $\mathit{Mgu}_X(\tau, \upsilon)$, where the type variables in $X$ are not bound in the resulting unifier. The implementation of this unification algorithm does not differ from previous presentations, such as Jones's unification algorithm for FCP [14]. We return any predicates in the type scheme of $x$, the updated substitution, and the observation that $x$ has been used.

The application case demonstrates the sets of used variables. We check the subexpressions $M$ and $N$; to account for the overloading of functions, we only assume that $M$ has type $u_1\, u_2\, \tau$, for some function type $u_1$. The variables used in $M$ are captured by $\Sigma$, and those used in $N$ are captured by $\Sigma'$. Any variables used in both must be unrestricted, and so the predicates inferred for the application include not just the predicates inferred for each sub expression ($P$ and $P'$), but also that any variables used in $\Sigma \cap \Sigma'$ must have unrestricted type. We capture this by $\mathit{Un}(\Gamma|_{\Sigma \cap \Sigma'})$, where $\Gamma|_\Sigma$ denotes the restriction of $\Gamma$ to variables in $\Sigma$. We give a declarative specification of $\mathit{Un}(-)$; an implementation that finds the simplest such $P$ can be straightforwardly derived from the definitions of $P \vdash -\ \mathsf{un}$ and entailment.

The `let` case demonstrates the treatment of polymorphism and binders. First, we must account for the possibility that $x$ was not used in $N$, and thus must be of unrestricted type. This is captured by $\mathit{Weaken}(x, \sigma, \Sigma)$. Second, we consider generalization. Recall the term $(\lambda x.x)\, y$, where $y$ has type $\tau$. The algorithm will infer that this term has type $\tau$ under the assumption $\mathsf{Fun}\ u$ for some variable $u$. But $u$ appears neither in the typing environment nor in the result, so naively generalizing this expression would give the (apparently ambiguous) type scheme $(\mathsf{Fun}\ u) \Rightarrow \tau$. However, this is not a real ambiguity: we have no way of observing the choice of $u$ in the resulting expression, so we could assume it to be $\overset{\circ}{\to}$ without decreasing the expressiveness or safety of type inference. We formalize this observation using an adaption of Jones's notion of improvement for qualified types [16]. An improving substitution for a qualified type $P \Rightarrow \tau$ is a substitution $S$ such that any satisfiable instance of $P \Rightarrow \tau$ is also a unambiguous satisfiable instance of $S\,(P \Rightarrow \tau)$. For example, $[\overset{\bullet}{\to}/f]$ is an improving substitution for $(\mathsf{Un}\, f, \mathsf{Fun}\, f) \Rightarrow \tau$, as the only ways to prove $\mathsf{Fun}\, f$ are if $f$ is $\overset{\bullet}{\to}$ or $\overset{\circ}{\to}$ and only the former is unrestricted. In the type $(\mathsf{Fun}\, f) \Rightarrow \tau$, where $f$ is not free in $\tau$ or the environment, we can instantiate $f$ to either $\overset{\bullet}{\to}$ or $\overset{\circ}{\to}$ and cannot observe the choice. As this choice does not introduce ambiguity, we consider $[\overset{\circ}{\to}/f]$ to be an improving substitution in such cases. We say that $S$ is an improving substitution for $X$ in $P$ if $S$ is the union of such improvements for each variable in $X$, and apply such an improving substitution before generalizing. Again, we give a declarative specification of $\mathit{GenI}(-,-)$, as the derivation of its implementation is entirely straightforward.

We can now relate type inference and the syntax-directed type system. First, inference constructs valid typings.

**Theorem 6** (Soundness of $\mathcal{M}$)**.** *If* $\mathcal{M}(S, X; \Gamma \vdash M : \tau) = P, S', \Sigma$, *then* $S'\, P \mid S'\, (S\, \Gamma) \vdash M : S'\, (S\, \tau)$.

The proof is by induction on the structure of $M$; each case involves comparing the predicates generated in inference to the predicates needed for typing. In combination with Theorem 3, this gives a similar soundness result for inference with respect to the original

$$\boxed{\mathcal{M}(S, X; \Gamma \vdash M : \tau) = P, S', \Sigma}$$

$$\mathcal{M}(S, X; \Gamma \vdash x : \tau) = ([u_i/t_i]\, P), U \circ S, \{x\}$$
$$\text{where } (x : \forall \vec{t}.P \Rightarrow \upsilon) \in S\,\Gamma$$
$$U = Mgu_X([u_i/t_i]\upsilon, \tau)$$

$$\mathcal{M}(S, X; \Gamma \vdash \lambda x.M : \tau) = Q, S', \Sigma \setminus x$$
$$\text{where } P; S'; \Sigma = \mathcal{M}(Mgu_X(\tau, u_1\, u_2\, u_3) \circ S, X; \Gamma, x : u_2 \vdash M : u_3)$$
$$Q = \mathsf{Fun}\, u_1 \cup Leq(u, \Gamma|_\Sigma) \cup Weaken(x, u, \Sigma)$$

$$\mathcal{M}(S, X; \Gamma \vdash M\,N : \tau) = Q, R', \Sigma \cup \Sigma'$$
$$\text{where } P, R, \Sigma = \mathcal{M}(S, X; \Gamma \vdash M : u_1\, u_2\, \tau)$$
$$P', R', \Sigma' = \mathcal{M}(R, X; \Gamma \vdash N : u_2)$$
$$Q = P \cup P' \cup \{\mathsf{Fun}\, u_1\} \cup Un(\Gamma|_{\Sigma\cap\Sigma'})$$

$$\mathcal{M}(S, X; \Gamma \vdash \mathsf{in}_i\, M : \tau) = P, R, \Sigma$$
$$\text{where } P, R, \Sigma = \mathcal{M}(Mgu_X(\tau, u_1 \oplus u_2) \circ S, X; \Gamma \vdash M : u_i)$$

$$\mathcal{M}(S, X; \Gamma \vdash \underline{\mathtt{case}}\, M\, \underline{\mathtt{of}}\, \{\mathsf{in}_1\, x \mapsto N;\ \mathsf{in}_2\, y \mapsto N'\} : \tau) = (P_M, P_N, P_{N'}, Q), R'', \Sigma_M \cup \Sigma_N \cup \Sigma_{N'}$$
$$\text{where } P_M, R, \Sigma_M = \mathcal{M}(S, X; \Gamma \vdash M : u_1 \oplus u_2)$$
$$P_N, R', \Sigma_N = \mathcal{M}(R, X; \Gamma, x : u_1 \vdash N : \tau)$$
$$P_{N'}, R'', \Sigma_{N'} = \mathcal{M}(R', X; \Gamma, y : u_2 \vdash N' : \tau)$$
$$\Sigma' = (\Sigma_N \setminus \Sigma_{N'}) \cup (\Sigma_{N'} \setminus \Sigma_N) \cup \Sigma_M \cap (\Sigma_N \cup \Sigma_{N'})$$
$$Q = Un(\Gamma|_{\Sigma'}) \cup Weaken(x, u, \Sigma_N) \cup Weaken(y, u', \Sigma_{N'})$$

$$\mathcal{M}(S, X; \Gamma \vdash \underline{\mathtt{let}}\, x = M\, \underline{\mathtt{in}}\, N : \tau) = (P', Q), R', \Sigma \cup (\Sigma' \setminus x)$$
$$\text{where } P, R, \Sigma = \mathcal{M}(S, X; \Gamma \vdash M : u_1)$$
$$\sigma = GenI(R\,\Gamma, R\,(P \Rightarrow u_1))$$
$$P', R', \Sigma' = \mathcal{M}(S, X; \Gamma, x : \sigma \vdash N : \tau))$$
$$Q = Un(\Gamma|_{\Sigma\cap\Sigma'}) \cup Weaken(x, \sigma, \Sigma')$$

$$\mathcal{M}(S, X; \Gamma \vdash K\,M : \tau) = P, R \circ U, \Sigma$$
$$\text{where } K : \forall \vec{t}.(\forall \vec{t'}.\exists \vec{t''}.\upsilon') \xrightarrow{\bullet} \upsilon$$
$$U = Mgu_X(\upsilon, [u_i/t_i]\tau)$$
$$P, R, \Sigma = \mathcal{M}(U \circ S, X \cup \vec{t'}; \Gamma \vdash M : [u_i/t_i, u_i''/t_i'']\upsilon')$$
$$t_i' \notin ftv((R \circ U)\,\Gamma)$$

$$\mathcal{M}(S, X; \Gamma \vdash \underline{\mathtt{let}}\, K\,x = M\, \underline{\mathtt{in}}\, N : \tau) = Q, R', \Sigma \cup (\Sigma' \setminus x)$$
$$\text{where } K : \forall \vec{t}.(\forall \vec{t'}.\exists \vec{t''}.\upsilon') \xrightarrow{\bullet} \upsilon$$
$$P_M, R, \Sigma_M = \mathcal{M}(S, X; \Gamma \vdash M : \upsilon)$$
$$P_N, R,' \Sigma_N = \mathcal{M}(R, X \cup \vec{t''}; \Gamma, x : [u_i/t_i, u_i'/t_i']\upsilon' \vdash N : \tau)$$
$$t_i'' \notin ftv(R'\,\Gamma, R'\,\tau)$$
$$Q = P_M \cup P_N \cup Weaken(x, \sigma, \Sigma_N) \cup Un(\Gamma|_{\Sigma_M\cap\Sigma_N})$$

$$\boxed{Leq, Un, Weaken, GenI}$$

$$Leq(\phi, \Gamma) = \bigcup\{P \mid P \vdash \phi \le \tau\}$$

$$Un(\Gamma) = \bigcup\{P \mid (y : \sigma) \in \Gamma, P \vdash \sigma\ \mathsf{un}\}$$

$$Weaken(x, \sigma, \Sigma) = \begin{cases} P & \text{if } x \notin \Sigma, P \vdash \sigma\ \mathsf{un} \\ \emptyset & \text{otherwise} \end{cases}$$

$$GenI(\Gamma, P \Rightarrow \tau) = \forall(ftv(S\,P, \tau)).S\,P \Rightarrow \tau$$
$$\text{where } S \text{ improves } ftv(P) \setminus ftv(\Gamma, \tau) \text{ in } P$$

**Figure 5:** Type inference algorithm $\mathcal{M}$. We let $u_i$ range over free variables, and write $[\tau_i/t_i]$ to denote the substitution $[\tau_1/t_1, \ldots, \tau_n/t_n]$.

type system. Next, we want to show that any valid typing can be found by inference.

**Theorem 7** (Completeness of $\mathcal{M}$)**.** *If $S$ is a substitution and $X$ is a set of type variables such that $P \mid S\,\Gamma \overset{S}{\vdash} M : S\,\tau$, and $S|_X = id$, then $\mathcal{M}(id, X; \Gamma \vdash M : \tau) = Q, S', \Sigma$ such that $(P \Rightarrow S\,\tau) \sqsubseteq GenI(S'\,\Gamma, S'\,Q \Rightarrow S'\,\tau)$.*

The proof is by induction on the typing derivation, observing in each case that the computed type generalizes the type in the derivation. Again, in combination with Theorem 5, we have a completeness result for inference with respect to the original type system. Finally, this allows us to give a constructive proof that Quill enjoys principal types.

**Theorem 8** (Principal Types)**.** *If $P_0 \mid H \vdash M : \sigma_0$ and $P_1 \mid H \vdash M : \sigma_1$ then there is some $\sigma$ such that $\emptyset \mid H \vdash M : \sigma$ and $(P_0 \mid \sigma_0) \sqsubseteq \sigma, (P_1 \mid \sigma_1) \sqsubseteq \sigma$.*

The soundness of inference tells us that, if there are any typings (let alone two typings) for a term in an environment, then the inference algorithm will compute a typing for that term. The completeness of inference tells us that the computed type will be more general than the original types.

### 4.4 Conservativity of Typing

We have claimed that Quill is as expressiveness as functional languages without linearity. To formalize that claim, we will show that any expression typeable in OML, Jones's core calculus for qualified types [15], is also typeable in Quill.

OML is a Core ML-like language with qualified types. Its types and terms are pleasingly simple: the former contains functions, type variables, and qualified and quantified types, and the latter contains variables, applications, abstractions, and `let` (to introduce polymorphism). We do not give a full description of OML typing here, partly as it is so similar to Quill typing. In particular, as in Quill typing, OML has a syntax directed typing judgment $P \mid \Gamma \overset{S}{\vdash}_{OML} M : \tau$, where $P$ is a collection of predicates, $\Gamma$ an OML typing environment and $\tau$ an OML type.

The crux of our argument is that (by construction) the syntax-directed typing rules of Quill can each be seen as generalizations of the corresponding rules of OML. For example, rules $(\to \mathrm{I}^{\mathrm{S}})$ and $(\to \mathrm{E}^{\mathrm{S}})$ can introduce generalization over function types, a feature not present in OML. However, they need not do so; if all functions are unrestricted, $(\to \mathrm{I}^{\mathrm{S}})$ and $(\to \mathrm{E}^{\mathrm{S}})$ are elaborate restatements of the corresponding rules of OML. The remaining difference is in the treatment of variables: Quill may insist on predicates to capture their unrestricted use, where there are no corresponding predicates required by OML, but this will never cause a term to be ill-typed.

**Theorem 9.** *If $P \mid \Gamma \overset{S}{\vdash}_{OML} M : \tau$, then there is some $Q$ such that $Q \mid \Gamma \overset{S}{\vdash} M : \tau$, and $Q \Rightarrow P$.*

OML also has a sound and complete type inference algorithm, and principal types. Thus, we see that if OML type inference accepts a given term, then Quill type inference will also accept the term,

and in each case will compute its most general typing. We might hope to show the converse as well; however, we do not know of a non-linear core calculus that matches the exact features of Quill, including both qualified types and datatype-mediated first class polymorphism.

We do not suggest that terms are given the same types in each setting: for example, the function $\lambda x.\lambda y.y$ is given the type $\forall tu.t \to u \to u$ in OML, whereas it would given the type $\forall tu.\mathsf{Un}\ t \Rightarrow t \to u \to u$ in Quill. Similarly, sums must be taken as primitive in Quill (as their elimination form shares its environment), whereas they can be encoded in OML. Finally, as demonstrated in our earlier discussion of monads (§3.3), Quill may suggest more refined abstractions than are present in non-linear languages. Nevertheless, this result does show a strong connection between programming in Quill and programming in traditional functional languages, one which is not shared by other combinations of linear and functional programming.

## 5. Semantics

We motivated the discussion of linear type systems by considering examples like session types and mutable arrays, in which we wanted to avoid duplicating or discarding values of linear types. The Quill type system, however, only restricts the use of assumptions, and says nothing about the use of values directly. Further, Quill differs from other substructural calculi in several ways, including the use of overloading and the form of first-class polymorphism. In this section, we demonstrate that Quill assures that the use of values, not just of assumptions, is consistent with their typing. To do so, we define a natural semantics for Quill terms, annotated with the values introduced and eliminated in the course of evaluation. We can then show that any values used non-linearly have unrestricted type. Our approach is strongly inspired by that used by Mazurak et al. [20] to prove a similar property of their F° calculus.

We begin by defining a notion of values for Quill. Intuitively, we might expect values to be abstractions, sums of values, or constructors applied to values. However, our intended safety property requires a refined notion of value, which distinguishes different instances of syntactically-identical values. The top of Figure 6 gives an extended syntax of Quill, in which values are tagged with indices from some index set $Ix$. The semantics will tag new values with fresh indices, and we will then rely on the indices to establish identity when showing safety. We identify a subset of values, $LinV$, as linear:

$$LinV = \{V \in Value \mid \text{if } \vdash V : \tau, \text{ then } \emptyset \not\Rightarrow \mathsf{Un}\ \tau\}.$$

Our goal is to show that values in $LinV$ are neither duplicated nor discarded during evaluation. The contents of $LinV$ depend on the signatures of the constructors. For a simple example, suppose that we have some $K$ with signature $(\exists u.u) \xrightarrow{\bullet} T$. To show $P \Rightarrow \mathsf{Un}\ T$, we would have to show that $P \Rightarrow \mathsf{Un}\ u$ (where $u \notin ftv(P)$). This is clearly impossible, so $K\,V \in LinV$ for any value $V$. On the other hand $\lambda x.x$ is not in $LinV$, as it can be given unrestricted type.

The bottom of Figure 6 gives a natural semantics for Quill. The evaluation relation $M \Downarrow_E^I V$ denotes that $M$ evaluates to $V$; the annotations $I$ and $E$ are multisets of values, $I$ capturing all values introduced during the evaluation and $E$ capturing all values eliminated during the evaluation. Functions evaluate to themselves, but annotated with a fresh index. The only value introduced is the function, and no values are eliminated. The other introduction forms are similar, but must account for the evaluation of their subexpressions. We use call-by-value evaluation; as observed by Mazurak et al., call-by-name and call-by-need evaluation may result in discarding linearly typed values during evaluation. The values introduced in evaluating an application are those introduced in evaluating each

$$\begin{array}{rcl} Ix \ni j,k & & \\ Value \ni V,W & ::= & K^j\,V \mid \lambda^j x.M \mid \mathsf{in}_i^j\,V \\ M,N & ::= & V \mid \ldots \end{array}$$

$$\frac{j \text{ fresh}}{\lambda x.M \Downarrow_\emptyset^{\lambda^j x.M} \lambda^j x.M} \qquad \frac{M \Downarrow_E^I V \quad j \text{ fresh}}{K\,M \Downarrow_E^{I,K^j\,V} K^j\,V}$$

$$\frac{M \Downarrow_E^I \lambda^j x.M' \quad N \Downarrow_{E'}^{I'} V \quad [V/x]M' \Downarrow_{E''}^{I''} W}{M\,N \Downarrow_{E,E',E'',\lambda^j x.M'}^{I,I',I''} W}$$

$$\frac{M \Downarrow_E^I V \quad j \text{ fresh}}{\mathsf{in}_i\,M \Downarrow_E^{I,\mathsf{in}_i^j\,V} \mathsf{in}_i^j\,V} \qquad \frac{M \Downarrow_E^I V \quad [V/x]N \Downarrow_{E'}^{I'} W}{\underline{\mathtt{let}}\ x = M\ \underline{\mathtt{in}}\ N \Downarrow_{E,E'}^{I,I'} W}$$

$$\frac{M \Downarrow_E^I \mathsf{in}_i^j\,V \quad [V/x]N_i \Downarrow_{E'}^{I'} W}{\underline{\mathtt{case}}\,M\,\underline{\mathtt{of}}\ \{\mathsf{in}_1\,x \mapsto N_1;\ \mathsf{in}_2\,y \mapsto N_2\} \Downarrow_{E,E',\mathsf{in}_i^j\,v}^{I,I'} W}$$

**Figure 6:** Linearity-aware semantics for Quill.

of its subexpressions and in evaluating the substituted result of the application. The values eliminated are those eliminated in each hypothesis and the function itself. The `let` and `case` rules are similar.

We can now state our desired safety property. Intuitively, if $M \Downarrow_E^I V$, we expect that each linear value introduced during evaluation (that is, each $W \in I \cap LinV$) will appear either exactly once, either in $E$ or as a subexpression of the result $V$. For any expression $M$, we define $SExp(M)$ to be the subexpressions of $M$, defined in the predictable fashion, $Exp(M) = \{M\} \cup SExp(M)$, and $Val(M) = Exp(M) \cap Value$.

**Theorem 10** (Type safety)**.** *Let $M$ be a closed term such that $\vdash M : \forall t.P \Rightarrow \tau$ and $M \Downarrow_E^I V$. Let $E' = E \cup Val(V)$, and let $D = I \setminus E'$ (the values discarded during evaluation) and $C = E' \setminus I$ (the values copied during evaluation). Then $P \mid \emptyset \vdash V : \tau$, and $W \in D \cup C$ only if $W \notin LinV$.*

The proof is by induction over the structure of $M$. The key observation is that duplication and discarding can happen only as the result of substitution, and thus that that linearity of variables (i.e., assumptions) is enough to assure that only unrestricted values are duplicated. We believe this argument can be straightforwardly generalized to small-step semantics, again following Mazurak et al.

## 6. Extensions

We describe three extensions of Quill, showing the generality of this approach.

Quill has a linear type system, in which both contraction (duplication) and weakening (discard) are limited to unrestricted types. Several alternative substructural logics exist: relevant logics, for example, exclude weakening but not contraction, and affine logics exclude contraction but not weakening. Some systems, such as that of Ahmed et al. [1] and Gan et al. [8] provide linear, affine, relevant, and unrestricted types simultaneously. Finally, there have been several type systems that introduce similar partitioning of assumptions to control side-effects, starting from Reynolds' work on Idealized Algol [24] and continuing with modern work on bunched implication [21] and separation logic. We have focused on the linear case in particular because various examples, such as session types, require its restrictions on both contraction and weakening. Nevertheless, we believe the Quill approach would apply equally well in these other cases. For example, the type system we have given uses

a single predicate, $\mathsf{Un}\ \tau$, when assumptions are either duplicated or discarded. Instead, we could introduce different predicates for these cases, say $\mathsf{Dup}\ \tau$ and $\mathsf{Drop}\ \tau$. The predicate $\mathsf{Un}\ \tau$ would then be the conjunction of $\mathsf{Dup}\ \tau$ and $\mathsf{Drop}\ \tau$. As in the systems of Ahmed at al. and Gan et al., we would require four arrow types. However, the remainder of the Quill approach would adapt seamlessly. We could extend the $\mathsf{Fun}\ \tau$ predicate and the $\geq$ relation to include the new arrow types, and the resulting system would continue to enjoy principal types and complete type inference.

The treatment of functions differs from the other primitive types (like products and sums) because the linearity of a function from $\tau$ to $\upsilon$ cannot be determined from linearity of $\tau$ and $\upsilon$. A similar observation can be made of existential types. For example, suppose that we have two constructors with signatures $K_1 :: (\exists u.u) \xrightarrow{\bullet} T_1\ t$ and $K_2 :: (\exists u.\mathsf{Un}\ u \Rightarrow u) \xrightarrow{\bullet} T_2\ t$. Assumptions of type $T_1\ \tau$ will always be treated as linear, and assumptions of type $T_2\ \tau$ always unrestricted, regardless of the choice of $\tau$. We could view $T_1$ and $T_2$ as instances of a general $T$ pattern (i.e., as the satisfying instances of a predicate $T\ t$) just as we view $\xrightarrow{\bullet}$ and $\xrightarrow{\circ}$ as instances of a general $\rightarrow$ pattern. Following the approach taken for functions, we would extend the $\geq$ relation to include the $T$ types, asserting that $\cdot \Rightarrow \tau \geq T_1\ \upsilon$ and $\mathsf{Un}\ \tau \Rightarrow \tau \geq T_2\ \upsilon$. We would then introduce a generalized constructor $K :: (T\ t, u \geq t) \Rightarrow u \xrightarrow{\bullet} t$, and a generalized deconstructor $unK :: (T\ t, t \geq f) \Rightarrow t \rightarrow (\forall u.u \rightarrow r) \xrightarrow{f} r$. As the goal is generalizing over the predicate $\mathsf{Un}\ u$, the body of the deconstructor cannot rely on its presence. This example demonstrates the flexibility of Quill; in particular, it shows that our treatment of functions is an instance of a more general pattern, itself expressible in Quill. We suspect that this treatment of existentials would also come closest to capturing how the use of existentials in non-linear functional languages could be expressed in linear calculi. On the other hand, there are cases for which this approach would not be appropriate, such as the use of existentials to enforce linear use of unrestricted primitives. We believe that more practical experience would be require to determine how, and how often, this view of existentials should be applied.

We have treated the `dup` and `drop` methods as providing a helpful intuition for the use of the `Un` predicate, but not actually themselves of interest. However, there are cases in which providing non-trivial implementations of these methods could be useful. For example, many operating system resources, such as file handles, need to be explicitly freed. One could imagine capturing such resources as affine types in a language based on Quill, in which the `drop` method freed the underlying resource. Similarly, given suitable primitives, one could imagine using `drop` and `dup` to implement a kind of reference-counting scheme for resources, in which `dup` incremented the reference count and `drop` decremented it. This approach would generalize the various scope based mechanisms for managing such resources in languages such as C# and Java. The derived definitions of `drop` and `dup` for products and sums (§3.1) would extend to this setting as well. However, this would introduce new concerns: the automatic insertion of the `drop` and `dup` methods. For a simple example, imagine that some variable $y$ is in scope in the expression $\lambda x.M$, but not free in $M$. We must insert a call to $\mathtt{drop}\ y$. Our current syntax-directed approach could be interpreted as moving calls to `drop` to the leaves of the typing derivation, but in this case that would delay the discard of $y$ until the function $\lambda x.M$ is invoked, which might be undesirable.

## 7. Related Work

The past thirty years have seen a wealth of work on linear types and their applications. We summarize some of the work most directly related to our own.

In introducing substructural type systems (§2.2), we described several other general purpose calculi, including F$^\circ$ of Mazurak et al. [20], and Alms of Tov and Pucella [27]. These systems were both influential on the development of Quill. Our work differs from theirs in two regards. First, we have generalized the treatment of functions, and thus the expressiveness of function combinators. We believe that, especially given the importance of combinator-based idioms in functional programming, this is a significant advance for the usability of linear functional calculi. Second, our treatment relies on qualified types, rather than building notions of subkinding, subtyping, and variance into the type system itself. While this may seem to simply be trading one kind of complexity for another, we believe that qualified types are an independently useful language feature (a claim borne out by the experience of Haskell). Finally, we believe that qualified types are a natural way to express relationships among types, as demonstrated by our generalization of relative linearity to existential types.

F$^\circ$, Alms, and Quill all rely on identifying a collection of types as unrestricted (through kind mechanisms in the first cases and type predicates in ours). There are several other mechanisms to integrate linear and unrestricted types. Wadler [29] and Barber and Plotkin [2] give calculi based directly on the propositions and proofs of linear logic, in which each linear type $\tau$ has an intuitionistic counterpart $!\tau$. These calculi also draw a distinction between intuitionistic and linear assumptions, where only the former are subject to contraction and weakening. While these calculi have close logical connections, the manipulation of the ! modality adds significant syntactic bureaucracy, and does not provide an obvious route to generalizing linear and unrestricted behavior. Gustavsson and Svenningsson [11] describe a system of usage annotations and bounded usage polymorphism; Walker [31] and Ahmed et al. [1] present similar system of annotations for linearity, albeit without the use of bounded polymorphism. These approaches seem less well suited for programming with linear types, however. For example, they provide linear Booleans (of little expressive value, as the duplication and discarding operations for Booleans can be easily defined) and unlimited session-typed channels (presumably an empty type). Finally, they require all types to be annotated with linearity (or usage) annotations, which is acceptable in a core language but unsuited to languages used by humans. Clean adopts a similar annotation-based approach with its uniqueness typing system [25]. However, the aims of uniqueness typing and linearity are dual: in Clean, a unique value can become non-unique (potentially at the cost of some of its operations), while with linear types we seek to guarantee linearity.

Finally, there have been numerous substructural approaches to typing for imperative and low-level languages, including region types [32], alias types [26], and adoption and focus [6], and several generalizations of linear typing, including coeffect systems [22]. These approaches have similar goals to our work—establishing safety guarantees beyond those expressed in traditional type systems—but differ in their underlying calculi and do not share our focus on principality and type inference. Nevertheless, some of the ideas of these systems could be profitably applied in ours. For example, some adaptation of the adoption and focus mechanisms could avoid the rebinding present in cases such as our binary session types example. We think exploring the overlap of our system and the problems they address, such as exploring explicit memory management in a Quill-like language, will be important future work.

## References

[1] A. J. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005,*

*Tallinn, Estonia, September 26-28, 2005*, pages 78–91, 2005.

[2] A. Barber and G. Plotkin. Dual intuitionistic linear logic. Technical Report LFCS-96-347, University of Edinburgh, 1996.

[3] J. Boyland. Checking interference with fractional permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 55–72, 2003.

[4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*. Springer, 2010.

[5] C. Chen and P. Hudak. Rolling your own MADT - A connection between linear types and monads. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 54–66, 1997.

[6] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 13–24, 2002.

[7] A. Filinski. Linear continuations. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 27–38, 1992.

[8] E. Gan, J. A. Tov, and G. Morrisett. Type classes for lightweight substructural types. In *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014.*, pages 34–48, 2014.

[9] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[10] J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[11] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In M. Mohnen and P. W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, volume 2011 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2000.

[12] J. C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 333–343, 1990.

[13] K. Honda. Types for dyadic interaction. In *CONCUR*. Springer, 1993.

[14] M. P. Jones. First-class polymorphism with type inference. In P. Lee, F. Henglein, and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 483–496. ACM Press.

[15] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[16] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 160–169, La Jolla, California, USA, 1995. ACM.

[17] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[18] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, 1998.

[19] S. Lindley and J. G. Morris. A semantics for propositions as sessions. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[20] K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in System F°. In *TLDI*, 2010.

[21] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[22] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: a calculus of context-dependent computation. In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135. ACM, 2014.

[23] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 25–36. ACM, 2008.

[24] J. C. Reynolds. Syntactic control of interference. In A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 39–46. ACM Press, 1978.

[25] S. Smetsers, E. Barendsen, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In H. J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany, January 1993, Proceedings*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer, 1993.

[26] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

[27] J. A. Tov and R. Pucella. Practical affine types. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458. ACM, 2011.

[28] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[29] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, pages 185–210, 1993.

[30] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[31] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1. The MIT Press, 2004.

[32] D. Walker, K. Crary, and J. G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.

## A. Packaging Unrestricted Channels

We might want to express session typing by wrapping an underlying unrestricted implementation of untyped channels (which we will call Chan, patterned on the Haskell Chan type). Previous work has demonstrated the use of existential types in doing this kind of wrapping. We have two problems:

1. How to capture the linearity of sessions, while still allowing End channels to be unrestricted; and,
2. How to capture the types of sent and received values

Ideally, we would like a solution that accomplishes both using existential types, avoiding the need for any waffle about the module system. Solving the first is actually relatively easy, and just relies on existing classes and simple existential types:

```
instance Un End
data Ch s = c ≥ s ⇒
            PackCh c (Dynamic → c → M c)
                     (c → M (Dynamic, c))

makeChannel :: Chan Dynamic → Ch s
makeChannel c =
  PackCh c (\v c → do writeChan c v; return c)
           (\c → do v ← readChan c; return (v, c))


send :: t ≥ f ⇒ t → Ch (t :!: s) -f-> M (Ch s)
send v (Ch c sender receiver) =
  do c ← sender (toDyn v) c
     return (Ch c sender receiver)

receive :: Ch (t :?: s) → M (t, Ch s)
receive (Ch c sender receiver) =
  do (v, c) ← receiver c
     return (fromDyn undefined v, Ch c sender receiver)
```

However, while this captures the linearity, it does not capture the typing. In particular, code with access to the Ch type may send a Dynamic value containing the wrong type, causing the corresponding fromDyn to fail. If we add in a notion of type equality I think we can do better, but at the cost of all the horrors of modern Haskell type systems. We start with type equality, which we can define using functional dependencies:

```
class t ≡ u | t → u, u → t
instance t ≡ t
```

We can then define the channel type as follows.

```
instance Un End
data PChan s = PChan (Chan Dynamic)
data Ch s = c s ≥ s ⇒
            PackCh (c s)
                   ((t ≥ f, s ≡ (t :!: s')) ⇒
                    t → c s → M (c s'))
                   ((s ≡ (t :?: s')) ⇒
                    c s → M (t, c s'))
```

The type PChan wraps an unrestricted channel with a phantom type variable. The type Ch follows the same pattern as before, but now encoding the form of the send and receive functions in the packaged sender and receiver. Consequently, this version depends on both first-class existentials and universals. Correspondingly, the implementations move the introduction and elimination of the Dynamic type into the packaged functions, but are otherwise unchanged.

```
makeChannel :: Chan Dynamic → Ch s
makeChannel c =
  PackCh (PChan c)
         (\v (PChan c) → do writeChan c (toDyn v)
                            return (PChan c))
         (\(PChan c) → do v ← readChan c
                          return (fromDyn undefined v,
                                  PChan c))


send :: t ≥ f ⇒ t → Ch (t :!: s) -f-> M (Ch s)
send v (Ch c sender receiver) =
  do c ← sender v c
     return (Ch c sender receiver)

receive :: Ch (t :?: s) → M (t, Ch s)
receive (Ch c sender receiver) =
  do (v, c) ← receiver c
     return (v, Ch c sender receiver)
```

## B. Encoding of Products in Quill

Additive products:

$$E(\tau \,\&\, \upsilon) = (\tau \geq f, \upsilon \geq f) \Rightarrow ((E(\tau) \xrightarrow{f} t) \oplus (E(\upsilon) \xrightarrow{f} t)) \xrightarrow{f} t$$

$$E([M,N]) = \lambda l.\underline{\mathsf{case}}\, l\, \underline{\mathsf{of}}\ \{\mathsf{in}_1 f \mapsto f\, E(M);\ \mathsf{in}_2\, g \mapsto g\, E(N)\}$$

$$E(\underline{\mathtt{fst}}\, M) = E(M)\,(\mathsf{in}_1\, id)$$

$$E(\underline{\mathsf{snd}}\, M) = E(M)\,(\mathsf{in}_2\, id)$$

Multiplicative products:

$$E(\tau \otimes \upsilon) = (E(\tau) \to E(\upsilon) \to t) \to t$$

$$E((t,u)) = \lambda f.ftu$$

$$E(\underline{\mathtt{let}}\ (x,y) = M\ \underline{\mathtt{in}}\ N) = M\,(\lambda x.\lambda y.N)$$