Idris 2: Quantitative Type Theory in Practice

Edwin Brady
School of Computer Science, University of St Andrews, Scotland, UK

Abstract

Dependent types allow us to express precisely *what* a function is intended to do. Recent work on Quantitative Type Theory (QTT) extends dependent type systems with *linearity*, also allowing precision in expressing *when* a function can run. This is promising, because it suggests the ability to design and reason about resource usage protocols, such as we might find in distributed and concurrent programming, where the state of a communication channel changes throughout program execution. As yet, however, there has not been a full-scale programming language with which to experiment with these ideas. Idris 2 is a new version of the dependently typed language Idris, with a new core language based on QTT, supporting linear and dependent types. In this paper, we introduce Idris 2, and describe how QTT has influenced its design. We give examples of the benefits of QTT in practice including: expressing which data is erased at run time, at the type level; and, resource tracking in the type system leading to type-safe concurrent programming with session types.

2012 ACM Subject Classification Software and its engineering → Functional languages

Keywords and phrases Dependent types, linear types, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.9

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.10`

Funding This work was funded by EPSRC grant EP/T007265/1.

Acknowledgements This work has benefitted from the many contributions to the Idris 2 project from the community. I am also grateful to the anonymous referees for their helpful feedback.

1 Introduction

Dependently typed programming languages, such as Idris [8], Agda [35], and Haskell with the appropriate extensions enabled [47], allow us to give precise types which can describe assumptions about and relationships between inputs and outputs. This is valuable for reasoning about *functional* properties, such as correctness of algorithms on collections [28], termination of parsing [14] and scope safety of programs [2]. However, reasoning about *non-functional* properties in this setting, such as memory safety, protocol correctness, or resource safety in general, is more difficult though it can be achieved with techniques such as embedded domain specific languages [9] or indexed monads [3, 27]. These are, nevertheless, heavyweight techniques which can be hard to compose.

Substructural type systems, such as linear type systems [45, 33, 6], allow us to express *when* an operation can be executed, by requiring that a linear resource be accessed *exactly once*. Being able to combine linear and dependent types would give us the ability to express an ordering on operations, as required by a protocol, with precision on exactly what operations are allowed, at what time. Historically, however, a difficulty in combining linear and dependent types has been in deciding how to treat occurrences of variables in *types*. This can be avoided [26] by never allowing types to depend on a linear term, but more recent work on Quantitative Type Theory (QTT) [4, 29] solves the problem by assigning a *quantity* to each binder, and checking terms at a specific *multiplicity*. Informally, in QTT, variables and function arguments have a multiplicity: 0, 1 or unrestricted (ω). We can freely use any variable in an argument with multiplicity 0 – e.g., in types – but we can not use a variable with multiplicity 0 in an argument with any other multiplicity. A variable bound with multiplicity 1 must be used exactly once. In this way, we can describe linear resource usage protocols, and furthermore clearly express *erasure* properties in types.

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller; Article No. 9; pp. 9:1–9:26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Idris 2 is a new implementation of Idris, which uses QTT as its core type theory. In this paper, we explore the possibilities of programming with a full-scale language built on QTT. By full-scale, we mean a language with high level features such as inference, interfaces, local function definitions and other syntactic sugar. As an example, we will show how to implement a library for concurrent programming with session types [21]. We choose this example because, as demonstrated by the extensive literature on the topic, correct concurrent programming is both hard to achieve, and vital to the success of modern software engineering. Our aim is to show that a language based on QTT is an ideal environment in which to implement accessible tools for software developers, based on decades of theoretical results.

1.1 Contributions

This paper is about exploring what is possible in a language based on Quantitative Type Theory (QTT), and introduces a new implementation of Idris. We make the following research contributions:

- We describe Idris 2 (Section 2), the first implementation of quantitative type theory in a full programming language, and the first language with full first-class dependent types implemented in itself.
- We show how Idris 2 supports two important applications of quantities: *erasure* (Section 3.2) which gives type-level guarantees as to which values are required at run-time, and *linearity* (Section 3.3) which gives type-level guarantees of resource usage. We also describe a general approach to implementing linear resource usage protocols (Section 4).
- We give an example of QTT in practice, encoding bidirectional session types (Section 5) for safe concurrent programming.

We do not discuss the metatheory of QTT, nor the trade-offs in its design in any detail. Instead, our interest is in discussing how it has affected the design of Idris 2, and in investigating the new kinds of programming and reasoning it enables. Where appropriate, we will discuss the intuition behind how argument multiplicities work in practice.

2 An Overview of Idris

Idris is a purely functional programming language, with *first-class* dependent types. That is, types can be computed, passed to and returned from functions, and stored in variables, just like any other value. In this section, we give a brief overview of the fundamental features which we use in this paper. A full tutorial is available online[1]. Readers who are already familiar with Idris may skip to Section 2.4 which introduces the new implementation.

2.1 Functions and Data Types

The syntax of Idris is heavily influenced by the syntax of Haskell. Function application is by juxtaposition and, like Haskell and ML and other related languages, functions are defined by recursive pattern matching equations. For example, to append two lists:

```
append : List a -> List a -> List a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

[1] https://idris2.readthedocs.io/en/latest/tutorial/index.html

The first line is a *type declaration*, which is required in Idris[2]. Names in the type declaration which begin with a lower case letter are *type-level variables*, therefore `append` is parameterised by an element type. Data types, like `List`, are defined in terms of their *type constructor* and *data constructors*:

```
data List : Type -> Type where
     Nil  : List elem
     (::) : elem -> List elem -> List elem
```

The type of types is `Type`. Therefore, this declaration states that `List` is parameterised by a `Type`, and can be constructed using either `Nil` (an empty list) or `::` ("cons", a list consisting of a head element and and tail). As we'll see in more detail shortly, types in Idris are first-class, thus the type of `List` (`Type -> Type`) is an ordinary function type. Syntax sugar allows us to write `[]` for `Nil`, and comma separated values in brackets expand to applications of `::`, e.g. `[1, 2]` expands to `1 :: 2 :: Nil`.

2.2 Interactive Programs

Idris is a pure language, therefore functions have no side effects. Like Haskell [37], we write interactive programs by *describing* interactions using a parameterised type `IO`. For example, we have primitives for console I/O, including:

```
putStrLn : String -> IO ()
getLine : IO String
```

`IO t` is the type of an interactive action which produces a result of type `t`. So, `getLine` is an interactive action which, when executed, produces a `String` read from the console. Idris the language *evaluates* expressions to produce a description of an interactive action as an `IO t`. It is the job of the run time system to *execute* the resulting action. Actions can be chained using the >>= operator:

```
(>>=) : IO a -> (a -> IO b) -> IO b
```

For example, to read a line from the console then echo the input:

```
echo : IO ()
echo = getLine >>= \x => putStrLn x
```

For readability, again like Haskell, Idris provides `do`-notation which translates an imperative style syntax to applications of >>=. The following definition is equivalent to the definition of `echo` above.

```
echo : IO ()
echo = do x <- getLine
          putStrLn x
```

The translation from `do`-notation to applications of >>= is purely syntactic. In practice therefore we can use `do`-notation in other contexts: for example, there is a `Monad` implementation for `IO`, and we will define an alternative >>= when implementing linear resource protocols.

[2] Note that unlike Haskell, we use a single colon for the type declaration.

2.3 First-Class Types

The main distinguishing feature of Idris compared to other languages, even some other languages with dependent types, is that types are *first-class*. For example we can pass them as arguments to functions, return them from functions, or store them in variables. This enables us to define type synonyms, to compute types from data, and express relationships between and properties of data. As an initial example, we can define a *type synonym*:

```
Point : Type
Point = (Int, Int)
```

Wherever the type checker sees `Point` it will evaluate it, and treat it as `(Int, Int)`:

```
moveRight : Point -> Point
moveRight (x, y) = (x + 1, y)
```

Languages often include type synonyms as a special feature (e.g. `typedef` in C or `type` declarations in Haskell). In Idris, no special feature is needed.

2.3.1 Computing Types

First-class types allow us to compute types from data. A well-known example is `printf` in C, where a format string determines the types of arguments to be printed. C compilers typically use extensions to check the validity of the format string; first-class types allow us to implement a `printf`-style variadic function, with compile time checking of the format. We begin by defining valid formats (limited to numbers, strings, and literals here):

```
data Format : Type where
     Num : Format -> Format
     Str : Format -> Format
     Lit : String -> Format -> Format
     End : Format
```

This describes the expected input types. We can calculate a corresponding function type:

```
PrintfType : Format -> Type
PrintfType (Num f) = (i : Int) -> PrintfType f
PrintfType (Str f) = (str : String) -> PrintfType f
PrintfType (Lit str f) = PrintfType f
PrintfType End = String
```

A function which computes a type can be used anywhere that Idris expects a value of type `Type`. So, for the type of `printf`, we name the first argument `fmt`, and use it to compute the rest of the type of `printf`:

```
printf : (fmt : Format) -> PrintfType fmt
```

We can check the type of an expression, even using an as yet undefined function, at the Idris REPL. For example, a format corresponding to `"%d %s"`:

```
Main> :t printf (Num (Lit " " (Str End)))
printf (Num (Lit " " (Str End))) : Int -> String -> String
```

We will implement this via a helper function which accumulates a string:

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
```

Idris has support for *interactive* development, via editor plugins and REPL commands, and we use *holes* extensively. An expression of the form `?hole` stands for an as yet unimplemented part of a program. This defines a top level function `hole`, with a type but no definition, which we can inspect at the REPL. So, we can write a partial definition of `printfFmt`:

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
printfFmt (Num f) acc = ?printfFmt_rhs_1
printfFmt (Str f) acc = ?printfFmt_rhs_2
printfFmt (Lit str f) acc = ?printfFmt_rhs_3
printfFmt End acc = ?printfFmt_rhs_4
```

Then, if we inspect the type of `printfFmt_rhs_1` at the REPL, we can see the types of the variables in scope, and the expected type of the right hand side:

```
Example> :t printfFmt_rhs_1
   f : Format
   acc : String
------------------------------
printfFmt_rhs_1 : Int -> PrintfType f
```

So, a format specifier of `Num` means we need to write a function that expects an `Int`. For reference, the complete definition is given in Listing 1. As a final step (omitted here) we can write a C-style `printf` by converting a `String` to a `Format` specifier. Here we use compile time data – the format specifier – to calculate the rest of the type. In Section 4, we will see a similar idea used to calculate a type from data which is not known until *run time*.

Listing 1 The complete definition of `printf`

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
printfFmt (Num f) acc = \i => printfFmt f (acc ++ show i)
printfFmt (Str f) acc = \str => printfFmt f (acc ++ str)
printfFmt (Lit str f) acc = printfFmt f (acc ++ str)
printfFmt End acc = acc

printf : (fmt : Format) -> PrintfType fmt
printf fmt = printfFmt fmt ""
```

2.3.2 Dependent Data Types

We define data types (such as `List a` and `Format` earlier) by giving explicit types for the *type constructor* and the *data constructors*. We are not limited to parameterising by types; type constructors can be parameterised by any value. The canonical example is a vector, `Vect`, a list with its length encoded in the type:

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a
```

`Nat` is the type of natural numbers, where `Z` stands for "zero" and `S` stands for "successor". It is represented by a machine integer at run time. As we noted in Section 2.1, lower case names in type definitions are type-level variables. So, when we define `append` on vectors...

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

. . . `n`, `a` and `m` are type-level variables. Note that we do not say "type variable" since they are not necessarily of type `Type`! Type-level variables are bound as *implicit arguments*. Written out in full, the type of `append` is:

```
append : {n : Nat} -> {m : Nat} -> {a : Type} ->
         Vect n a -> Vect m a -> Vect (n + m) a
```

Note: We will refine this when introducing multiplicities in Section 3.

The implicit arguments are concrete arguments to `append`, like the `Vect` arguments. Their values are solved by unification [30, 20]. Typically implicit arguments are only necessary at compile time, and unused at run time. However, we can use an implicit argument in a definition, since the names of the arguments are in scope in the right hand side:

```
length : {n : Nat} -> Vect n a -> Nat
length xs = n
```

One challenge with first-class types is in distinguishing those parts of programs which are required at run time, and those which can be erased. Tradtionally, this phase distinction has been clear: types are erased at run time, values preserved. But this correspondence is merely a coincidence, arising from the special (non first-class) status of types! As we see with `length` and `append`, sometimes an argument might be required (in `length`) and sometimes it might be erasable (in `append`). Idris 1 uses a constraint solving algorithm [41], which has been effective in practice, but has a weakness that it is not possible to tell from a definition's type alone which arguments are required at run time. In Section 3.2 we will see how quantitative type theory allows us to make a precise distinction between the run time relevant parts of a program and the compile time only parts.

2.4 Idris 2

Idris 2 is a new version of Idris, implemented in itself, and based on Quantitative Type Theory (QTT) as defined in recent work by Atkey [4] following initial ideas by McBride [29]. In QTT, each variable binding is associated with a *quantity* (or *multiplicity*) which denotes the number of times a variable can be used in its scope: either zero, exactly once, or unrestricted. We will describe these further shortly. Several factors have motivated the new implementation:

- In implementing Idris in itself, we have necessarily done the engineering required on Idris to implement a system of this scale. Furthermore, although it is outside the scope of the present paper, we can explore the benefits of dependently typed programming in implementing a full-scale programming language.
- A limitation of Idris 1 is that it is not always clear which arguments to functions and constructors are required *at run time*, and which are erased, even despite previous work [7, 41, 42]. QTT allows us to state clearly, in a type, which arguments are erased. Erased arguments are still relevant *at compile time*.
- There has, up to now, been no full-scale implementation of a language based on QTT which allows exploration of the possibilities of linear and dependent types.
- Purely pragmatically, Idris 1 has outgrown the requirements of its initial experimental implementation, and since significant re-engineering has been required, it was felt that it was time to re-implement it in Idris itself.

In the following sections, we will discuss the new features in Idris 2 which arise as a result of using QTT as the core: firstly, how to express *erasure* in the type system; and secondly, how to encode resource usage protocols using linearity.

3 Quantities in Types

The biggest distinction between Idris 2 and its predecessor is that it is built on Quantitative Type Theory (QTT) [29, 4]. In QTT, each variable binding (including function arguments) has a *multiplicity*. QTT itself takes a general approach to multiplicities, allowing any semiring. For Idris 2, we make a concrete choice of semiring, where a multiplicity can be one of:

- 0: the variable is not used at run time
- 1: the variable is used exactly once at run time
- ω: no restrictions on the variable's usage at run time

In this section, we describe the syntax and informal semantics of multiplicities, and discuss the most important practical applications: erasure, and linearity. The formal semantics are presented elsewhere [4]; here, we aim to describe the intuition. In summary:

- Variable bindings have multiplicities which describe how often the variable must be used within the scope of its binding.
- A variable is "used" when it appears in the body of a definition (that is, not a type declaration), in an argument position with multiplicity 1 or ω.
- A function's type gives the multiplicities the arguments have in the function's body.

Variables with multiplicity ω are truly unrestricted, meaning that they can be passed in argument positions with multiplicity 0, 1 or ω. A function which takes an argument with multiplicity 1 promises that it will not share the argument in the future; there is no requirement that it has not been shared in the past.

3.1 Syntax

When declaring a funcion type we can, optionally, give an explicit multiplicity of `0` or `1`. If an argument has no multiplicity given, it defaults to ω. For example, we can declare the type of an identity function which takes its polymorpic type *explicitly*, and mark it erased:

```
id_explicit : (0 a : Type) -> a -> a
```

If we give a partial definition of `id_explicit`, then inspect the type of the hole in the right hand side, we can see the multiplicities of the variables in scope:

```
id_explicit a x = ?id_explicit_rhs

Main> :t id_explicit_rhs
0 a : Type
  x : a
------------------------------
id_explicit_rhs : a
```

This means that `a` is not available at run time, and `x` is unrestricted at run time. If there is no explicit multiplicity shown, it is ω. A variable which is not available at run time can only be used in argument positions with multiplicity `0`. Implicitly bound arguments are also given multiplicity `0`. So, in the following declaration of `append`...

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

...`n`, `a` and `m` have multiplicity `0`. In Idris 2, therefore, unlike Idris 1, the declaration is equivalent to writing:

```
append : {0 n : Nat} -> {0 m : Nat} -> {0 a : Type} ->
         Vect n a -> Vect m a -> Vect (n + m) a
```

The default multiplicities for arguments are, therefore:

- If you explicitly write the binding, the default is ω.
- If you omit the binding (e.g. in a type-level variable), the default is 0.

This design choice, that type-level variables are by default erased, is primarily influenced by the common usage in Idris 1, that implicit type-level variables are typically compile-time only. As a result, the most common use cases involve the most lightweight syntax.

3.2 Erasure

The multiplicity `0` gives us control over whether a function argument – `Type` or otherwise – is used at run time. This is important in a language with first-class types since we often parameterise types by values in order to make relationships between data explicit, or to make assumptions explicit. In this section, we will consider two examples of this, and see how multiplicity 0 allows us to control which data is available at run time.

3.2.1 Example 1: Vector length

We have seen how a vector's type includes its length; we can use this length at run time, even though it is part of the type, provided that it has non-zero multiplicity:

```
length : {n : Nat} -> Vect n a -> Nat
length xs = n
```

With this definition, the length `n` is available at run time, since `{n : Nat}` is an explicitly written binding so has default multiplicity ω. This has a run time cost, in that `n` is passed to the function at run time, as well as potentially the cost of computing the length elsewhere. If we want the length to be erased, we would need to recompute it in the `length` function:

```
length : Vect n a -> Nat
length [] = Z
length (_ :: xs) = S (length xs)
```

The type of `length` in each case makes it explicit whether or not the length of the `Vect` is available. Let us now consider a more realistic example, using the type system to ensure soundness of a compressed encoding of a list.

3.2.2 Example 2: Run-length Encoding of Lists

Run-length encoding is a compression scheme which collapses sequences (runs) of the same element in a list. It was originally developed for reducing the bandwidth required for television pictures [39], and later used in image compression and fax formats, among other things.

We will define a data type for storing run-length encoded lists, and use the type system to ensure that the encoding is sound with respect to the original list. To begin, we define a function which constructs a list by repeating an element a given number of times. We will need this for explaining the relationship between compressed and uncompressed data.

```
rep : Nat -> a -> List a
rep Z x = []
rep (S k) x = x :: rep k x
```

Using this, and the concatenation operator for `List` (`++`, which is defined like `append`), we can describe what it means to be a run-length encoded list:

```
data RunLength : List ty -> Type where
     Empty : RunLength []
     Run : (n : Nat) -> (x : ty) -> (rle : RunLength more) ->
           RunLength (rep (S n) x ++ more)
```

- `Empty` is the run-length encoding of the empty list `[]`
- Given a length `n`, an element `x`, and an encoding `rle` of the list `more`, `Run n x rle` is the encoding of the list `rep (S n) x ++ more`. That is, $n + 1$ copies of `x` followed by `more`.

We use `S n` to ensure that `Run` always increases the length of the list, but otherwise we make no attempt (in the type) to ensure that the encoding is optimal; we merely ensure that the encoding is sound with respect to the encoded list. Let us try to write a function which uncompresses a run-length encoded list, beginning with a partial definition:

```
uncompress : RunLength {ty} xs -> List ty
uncompress rle = ?uncompress_rhs
```

Note: The `{ty}` syntax gives an explicit value for the implicit argument to `RunLength`. This means that the `ty` argument to RunLength, and hence the element type of `xs`, is the same type as the element type of the list returned by `uncompress`.

Like our initial implementation of `length` on `Vect`, we might be tempted to return `xs` directly, since the index of the encoded list gives us the original uncompressed list. However, if we check the type of `uncompress_rhs`, we see that `xs` has multiplicity 0 because it is not an explicit argument, so isn't available at run time:

```
 0 xs : List ty
   rle : RunLength xs
------------------------------
uncompress_rhs : List ty
```

This is a good thing: if the uncompressed list were available at run-time, there would have been no point in compressing it! We can still take advantage of having the uncompressed list available as part of the type, though, by refining the type of `uncompress`:

```
data Singleton : a -> Type where
     Val : (x : a) -> Singleton x

uncompress : RunLength xs -> Singleton xs
```

A value of type `Singleton x` has exactly one value, `Val x`. The type of `uncompress` expresses that the uncompressed list is guaranteed to have the same value as the original list, although it must still be reconstructed at run-time. We can implement it as follows:

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y) = let Val ys = uncompress y in
                             Val (x :: (rep n x ++ ys))
```

Aside: This implementation was generated by type-directed program synthesis [38], rather than written by hand, taking advantage of the explicit relationship given in the type between the input and the output. The definition of `RunLength` more or less directly gives the uncompression algorithm, so we should not need to write it again!

The `0` multiplicity, therefore, allows us to reason about values at compile time, without requiring them to be present at run time. Furthermore, QTT gives us a guarantee of erasure, as well as an explicit type-level choice as to whether an index is erased or not.

3.3 Linearity

An argument with multiplicity `0` is guaranteed to be erased at run time. Correspondingly, an argument with multiplicity `1` is guaranteed to be used exactly once. The intuition, similar to that of Linear Haskell [6], is that, given a function type of the form...

```
f : (1 x : a) -> b
```

...then, if an expression `f x` is evaluated exactly once, `x` is evaluated exactly once in the process. QTT is a new core language, and the combination of linearity with dependent types has not yet been extensively explored. Thus, we consider the multiplicity `1` to be experimental, and in general Idris 2 programmers can get by without it – nothing in the Prelude exposes an interface which requires it. Nevertheless, we have found that an important application of linearity is in controlling resource usage. In the rest of this section, we describe two examples of this. First, we show in general how linearity can prevent us duplicating an argument, which can be important if the argument represents an external resource; then, we give a more concrete example showing how the `IO` type described in Section 2.2 is implemented.

3.3.1 Example 1: Preventing Duplication

To illustrate multiplicity `1` in practice, we can try (and fail!) to write a function which duplicates a value declared as "use once", interactively, where `LPair` is a linear pair type:

```
dup : (1 x : a) -> LPair a a
dup x = ?dup_rhs
```

Inspecting the `dup_rhs` hole shows that we have:

```
 0 a : Type
 1 x : a
--------------------------------------
dup_rhs : LPair a a
```

So, `a` is not available at run-time, and `x` must be used exactly once in the definition of `dup_rhs`. We can write a partial definition, where `#` is the constructor of `LPair`:

```
dup x = x # ?second_x
```

However, if we check the hole `second_x` we see that `x` is not available, because there was only 1 and it has already been consumed:

```
 0 a : Type
 0 x : a
--------------------------------------
second_x : a
```

We see the same result if we try `dup x = (?second_x, x)`. If we persist, and try...

```
dup x = x # x
```

...then Idris reports "`There are 2 uses of linear name x`".

▶ Remark. As we noted earlier, only usages in the body of a definition count. This means we can still parameterise data by linear variables. For example, if we have an `Ordered` predicate on lists, we can write an `insert` function on ordered linear lists:

```
insert : a -> (1 xs: List a) -> (0 _ : Ordered xs) -> List a
```

The use in `Ordered` does not count, and since `Ordered` has multiplicity 0 it is erased at run time, so any occurrence of `xs` when building the `Ordered` proof also does not count.

3.3.2 Example 2: I/O in Idris 2

Like Idris 1 and Haskell, Idris 2 uses a parameterised type `IO` to describe interactive actions. Unlike the previous implementation, this is implemented via a function which takes an abstract representation of the outside world, of primitive type `%World`:

```
PrimIO : Type -> Type
PrimIO a = (1 x : %World) -> IORes a
```

The `%World` is consumed exactly once, so it is not possible to use previous worlds (after all, you can't unplay a sound, or unsend a network message). It returns an `IORes`:

```
data IORes : Type -> Type where
     MkIORes : (result : a) -> (1 w : %World) -> IORes a
```

This is a pair of a result (with usage ω), and an updated world state. The intuition for multiplicities in data constructors is the same as in functions: here, if `MkIORes result w` is evaluated exactly once, then the world `w` is evaluated exactly once. We can now define `IO`:

```
data IO : Type -> Type where
     MkIO : (1 fn : PrimIO a) -> IO a
```

There is a primitive `io_bind` operator (which we can use to implement >>=), which guarantees that an action and its continuation are executed exactly once:

```
io_bind : (1 act : IO a) -> (1 k : a -> IO b) -> IO b
io_bind (MkIO fn) = \k => MkIO (\w => let MkIORes x' w' = fn w
                                      MkIO res = k x' in res w')
```

The multiplicities of the `let` bindings are inferred from the values being bound. Since `fn w` uses `w`, which is required to be linear from the type of `MkIO`, `MkIORes x' w'` must itself be linear, meaning that `w'` must also be linear. This implementation of `IO` is similar to the Haskell approach [37], with two differences:

1. The `%World` token is guaranteed to be consumed exactly once, so there is a type-level guarantee that the outside world is never duplicated or thrown away.
2. There is no built-in mechanism for exception handling, because the type of `io_bind` requires that the continuation is executed exactly once. So, in `IO` primitives, we must be explicit about where errors can occur. One can, however, implement higher level abstractions which allow exceptions if required.

Linearity is, therefore, fundamental to the implementation of `IO` in Idris 2. Fortunately, none of the implementation details need to be exposed to application programmers who are using `IO`. However, once a programmer has an understanding of the combination of linear and dependent types, they can use it to encode and verify more sophisticated APIs.

4 Linear Resource Usage Protocols

The `IO` type uses a linear resource representing the current state of the outside world. But, often, we need to work with other resources, such as files, network sockets, or communication channels. In this section, we introduce an extension of `IO` which allows creating and updating linear resources, and show how to use it to implement a resource usage protocol for an automated teller machine (ATM).

4.1 Initialising Linear Values

In QTT, multiplicities are associated with *binders*, not with return values or types. This is a design decision of QTT, rather than of Idris, and has the advantage that we can use a type linearly or not, depending on context. We can write functions that create values to be used linearly by using *continuations*, for example, to create a new linear array:

```
newArray : (size : Int) -> (1 k : (1 arr : Array t) -> a) -> a
```

The array must be used exactly once in the scope of `k`, and if this is the only way of constructing an `Array`, then all arrays are guaranteed to be used linearly, so we can have in-place update. As a matter of taste, however, we may not want to write programs with explicit continuations. Fortunately, `do` notation can help; recall the bind operator for `IO`:

```
(>>=) : IO a -> (a -> IO b) -> IO b
```

This allows us to chain an IO action and its continuation, and `do` notation gives syntactic sugar for translating into applications of >>=. Therefore, we can define an alternative >>= operator for chaining actions which return linear values. We will achieve this by defining a new type for capturing interactive actions, extending `IO`, and defining its bind operator.

4.2 Linear Interactive Programs

First, we define how many times the result of an operation can be used. These correspond to the multiplicities 0, 1 and ω:

```
data Usage = None | Linear | Unrestricted
```

We declare a data type `L`, which describes interactive programs that produce a result with a specific multiplicity. We choose a short name `L` since we expect this to be used often:

```
data L : {default Unrestricted use : Usage} ->
         Type -> Type where
```

In Section 2.3.2 we described implicit arguments, which are solved by unification. Here, `use` is a *default implicit* argument, and the `default Unrestricted` annotation means that if its value is not given explicitly, it will take a default value of `Unrestricted`.

Like `IO`, `L` provides the operators `pure` and >>=. However, unlike `IO`, they need to account for variable usage. One limitation of QTT is that it does not yet support quantity polymorphism, so we must provide separate `pure` operators for each quantity:

```
pure  : (  x : a) -> L         a
pure0 : (0 x : a) -> L {use=0} a
pure1 : (1 x : a) -> L {use=1} a
```

Idris translates integer literals using the `fromInteger` function. We have defined a `fromInteger` function that maps `0` to `None` and `1` to `Linear` which allows us to use integer literals as the values for the `use` argument.

The type of >>= is more challenging. In order to take advantage of `do`-notation, we need a single >>= operator for chaining an action and a continuation, but there are several possible combinators of variable usage. Consider:

- The action might return an erased, linear or unrestricted value.
- Correspondingly, the continuation must bind its argument at multiplicity 0, 1 or ω.

In other words, the type of the continuation to >>= depends on the usage of the result of the action. We can therefore take advantage of first-class types and *calculate* the continuation type. Given the usage of the action (`u_a`), the usage of the continuation (`u_k`) and the return types of each, `a` and `b`, we calculate:

```
ContType : (u_a : Usage) -> (u_k : Usage) -> Type -> Type -> Type
ContType None u_k a b = (0 _ : a) -> L {use=u_k} b
ContType Linear u_k a b = (1 _ : a) -> L {use=u_k} b
ContType Unrestricted u_k a b = a -> L {use=u_k} b
```

Then, we can write a type for >>= as follows:

```
(>>=) : {u_a : _} ->
        (1 _ : L {use=u_a} a) ->
        (1 _ : ContType u_act u_k a b) -> L {use=u_k} b
```

The continuation type is calculated from the usage of the first action, and is correspondingly needed in the implementation, so `u_a` is run time relevant. However, in practice it is removed by inlining. Fortunately, the user of `L` need not worry about these details. They can freely use `do`-notation and let the type checker take care of variable usage for them.

Finally, for developing linear wrappers for `IO` libraries, we allow lifting `IO` actions:

```
action : IO a -> L a
```

We use `action` for constructing primitives. Note that we will not be able to bypass any linearity checks this way, since it does not promise to use the `IO` action linearly, so we cannot pass any linear resources to an `action`. The implementation of `L` is via a well-typed interpreter [5], a standard pattern in dependently typed programming.

Note: `L` is defined in a library `Control.Linear.LIO`, distributed with Idris 2. In the library, its type is more general; `L : (io : Type -> Type) -> {use : Usage} -> Type -> Type`. This allows us to extend *any* monad with linearity, not just `IO`, but this generality is not necessary for the examples in this paper.

4.3 Example: An ATM State Machine

We can use linear types to encode the state of a resource, and implement operations in `L` to ensure that they are only executed when the resource is in the appropriate state. For example, an ATM should only dispense cash when a user has inserted their card and entered a correct PIN. This is a typical sequence of operations on an ATM:

- A user inserts their bank card.
- The machine prompts the user for their PIN, to check the user is entitled to use the card.
- If PIN entry is successful, the machine prompts the user for an amount of money, and then dispenses cash.

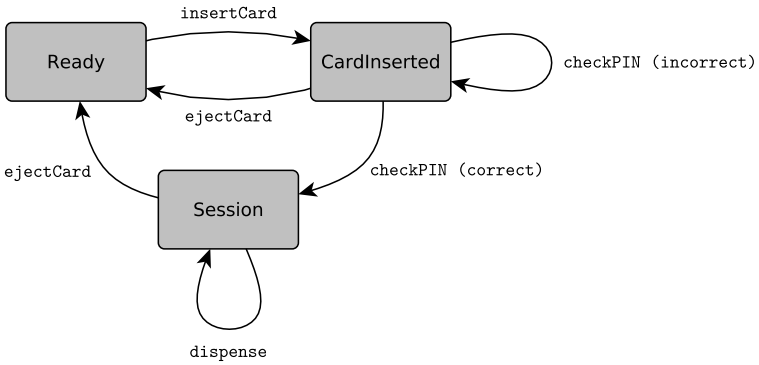

Figure 1 A state machine describing the states and operations on an ATM.

Figure 1 defines, at a high level, the states and operations on an ATM, showing when each operation is valid. We will define these operations as functions in the `L` type, using a linear reference to a representation of an ATM. We define the valid states of the ATM as a data type, then an ATM type which is parameterised by its current state, which is one of:

- `Ready`: the ATM is ready and waiting for a card to be inserted.
- `CardInserted`: there is a card inside the ATM but the PIN entry is not yet verified.
- `Session`: there is a card inside the ATM and the PIN has been verified.

```
data ATMState = Ready | CardInserted | Session
data ATM : ATMState -> Type
```

We leave the definition of `ATM` abstract. In practice, this is where we would need to handle implementation details such as how to access and update a user's bank account. For the purposes of this example, we are only interested in encoding the high level state transitions in types. We will need functions to initialise and shut down the reference:

```
initATM  : L {use=1} (ATM Ready)
shutDown : (1 _ : ATM Ready) -> L ()
```

`initATM` creates a linear reference to an ATM in the initial state, `Ready`, which must be used exactly once. Correspondingly, `shutDown` deletes the linear reference. Listing 2 presents the types of the remaining operations, implementing the transitions from Figure 1.

We have: user-directed state transitions, where the *programmer* is in control over whether an operation succeeds; general purpose operations, which do not change the state, and are part of the machine's user interface; and, machine-directed state transitions, where the *machine* is in control over whether an operation succeeds, for example the machine decides if PIN entry was correct.

User-directed state transitions

The `insertCard` card function takes a machine in the `Ready` state, and returns a new machine in the `CardInserted` state. The type ensures that we can only run the function with the machine in the appropriate state. For `dispense`, we need to satisfy the security property that the machine can only dispense money in a validated session. Thus, it has an input state of `Session`, and the session remains valid afterwards.

Listing 2 Operations on an ATM

```
data HasCard : ATMState -> Type where
     HasCardPINNotChecked : HasCard CardInserted
     HasCardPINChecked : HasCard Session

data PINCheck = CorrectPIN | IncorrectPIN

insertCard : (1 _ : ATM Ready) -> L {use=1} (ATM CardInserted)
checkPIN : (1 _ : ATM CardInserted) -> (pin : Int) ->
           L {use=1}
             (Res PINCheck
                  (\res => ATM (case res of
                                     CorrectPIN => Session
                                     IncorrectPIN => CardInserted)))
dispense : (1 _ : ATM Session) -> L {use=1} (ATM Session)
getInput : HasCard st => (1 _ : ATM st) ->
                         L {use=1} (Res String (const (ATM st)))
ejectCard : HasCard st => (1 _ : ATM st) -> L {use=1} (ATM Ready)
message : (1 _ : ATM st) -> String -> L {use=1} (ATM st)
```

For `ejectCard`, it is only valid to eject the card if there is already a card in the machine. This is true in *two* states: `CardInserted` and `Session`. Therefore, we define a *predicate* on states which holds for states where there is a card in the machine:

```
data HasCard : ATMState -> Type where
     HasCardPINNotChecked : HasCard CardInserted
     HasCardPINChecked : HasCard Session
```

The type of `ejectCard` then takes an input of type `HasCard st`, which is a proof that the predicate holds for the machine's input state `st`.

```
ejectCard : HasCard st => (1 _ : ATM st) -> L {use=1} (ATM Ready)
```

The notation `HasCard st => ...`, with the `=>` operator, means that this is an *auto implicit* argument. Like implicits, and `default` implicits, these can be omitted. The type checker will attempt to fill in a value by searching the possible constructors. In this case, if `st` is `CardInserted`, the value will be `HasCardPINNotCheck`, and if it is `Session`, the value will be `HasCardPINChecked`. Otherwise, Idris will not be able to find a value, and will report an error. Auto implicits are also used for interfaces, corresponding to type classes in Haskell, although we do not use interfaces elsewhere in this paper.

General purpose operations

The `message` function displays a message to the user. Its type means that we can display a message no matter the machine's state. Nevertheless, since an ATM is linear, we must return a new reference. The `getInput` function reads input from the user, using the machine's keypad, provided that there is a card in the machine. Again, this needs to return a new reference, along with the input. `Res` is a *dependent pair* type, where the first item is unrestricted, and the second item is linear with a type computed from the value of the first element. We describe this below in the context of `checkPIN`.

Machine-directed state transitions

The most interesting case is `checkPIN`. In the other functions, the programmer is in control of when state transitions happen, but in this case, the transition may or may not succeed, depending on whether the PIN is correct. To capture this possibility, we return the result in a dependent pair type, `Res`, defined as follows in the Idris Prelude:

```
data Res : (a : Type) -> (a -> Type) -> Type where
     (#) : (val : a) -> (1 r : t val) -> Res a t
```

This pairs a value, `val`, with a linear resource whose type is computed from the value. This can be illustrated with a partial ATM program:

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            ok # m <- checkPIN m 1234
            ?whatnow
```

This program initialises an ATM, inserts a card, then checks whether the card has the PIN 1234. Checking the PIN returns a `Res`, which we deconstruct, then we can inspect the hole `?whatnow` to see where to go from here:

```
   ok : PINCheck
 1 m : ATM (case ok of { CorrectPIN => Session
                         IncorrectPIN => CardInserted })
------------------------------
whatnow : L ()
```

So, we have the result of the `PINCheck`, and an updated `ATM`, but we will only know the state of the `ATM`, and hence be able to make progress in the protocol, if we actually check the returned result! We cannot know statically what the next state of the machine is going to be, but using first class types, we *can* statically check that the necessary dynamic check is made. We could also use a sum type, such as `Either`, for the result of the PIN check, e.g.

```
checkPIN : (1 _ : ATM CardInserted) -> (pin : Int) ->
           L {use=1} (Either (ATM CardInserted) (ATM Session))
```

This would arguably be simpler. On the other hand, by returning an `ATM` with an as yet unknown state, we can still run operations on the `ATM` such as `message` even before resolving the state. This might be useful for diagnostics, or for user feedback, for example. Listing 3 shows one possible ATM protocol implementation, displaying a message before checking the PIN, then dispensing cash if the PIN was valid. Note that the protocol also requires the card to have been ejected before the machine is shut down.

Aside: Linearity and exceptions do not mix well, since when we catch an exception, we need to know what state the machine was in at the point it was thrown in order to clean up effectively. On the other hand, if we have to check every result as in Listing 3, we will end up with a lot of nested `case` blocks, which are hard to read. As a compromise, Idris provides a pattern matching bind notation [10], which allows us to code to a "happy path" and deal with alternatives as they arise. For example:

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            CorrectPIN # m <- checkPIN m 1234
               | IncorrectPIN # m => ?failure
            ?success
```

Listing 3 Example ATM protocol implementation

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            ok # m <- checkPIN m 1234
            m <- message m "Checking PIN"
            case ok of
                 CorrectPIN => do m <- dispense m
                                  m <- ejectCard m
                                  shutDown m
                 IncorrectPIN => do m <- ejectCard m
                                    shutDown m
```

The "happy path" is that the PIN was entered correctly. The alternative we need to handle is the `IncorrectPIN` case, which we can handle in a similar manner to Listing 3.

5 Session Types via QTT

To illustrate how we can use quantities on a more substantial example, let us consider how to use them to implement session types. Session types [21, 22] give types to communication channels, allowing us to express exactly *when* a message can be sent on a channel, ensuring that communication protocols are implemented completely and correctly. There has been extensive previous work on defining calculi for session types[3]. In Idris 2, the combination of linear and dependent types means that we can implement session types directly:

- **Linearity** means that a channel can only be accessed once, and once a message has been sent or received on a channel, the channel is in a new state.
- **Dependent Types** give us a way of describing protocols at the type level, where progress on a channel can change according to values sent on the channel.

A complete implementation of session types would be a paper in itself, so we limit ourselves to dyadic session types in concurrent communicating processes. We assume that functions are *total*, so processes will not terminate early and communication will always succeed. In a full library, dealing with *distributed* as well as *concurrent* processes, we would also need to consider failures such as timeouts and badly formed messages [18].

The key idea is to parameterise channels by the actions which will be executed on the channel – that is, the messages which will be sent and received – and to use channels linearly. We declare a `Channel` type as follows:

```
data Actions : Type where
     Send : (a : Type) -> (a -> Actions) -> Actions
     Recv : (a : Type) -> (a -> Actions) -> Actions
     Close : Actions

data Channel : Actions -> Type
```

[3] A collection of implementations is available at `http://groups.inf.ed.ac.uk/abcd/session-implementations.html`

Internally, `Channel` contains a message queue for bidirectional communication. Listing 4 shows the types of functions for initiating sessions, and sending and receiving messages. In the type of `send`, we see that to send a value of type `ty` we must have a channel in the state `Send ty next`, where `next` is a function that computes the rest of the protocol. The type of `recv` shows that we compute the rest of the protocol by inspecting the value received. We initiate concurrent sessions with `fork`, and will discuss the details of this shortly.

Listing 4 Initiating and executing concurrent sessions

```
send : (1 chan : Channel (Send ty next)) -> (val : ty) ->
       L {use=1} (Channel (next val))
recv : (1 chan : Channel (Recv ty next)) ->
       L {use=1} (Res ty (\val => Channel (next val)))
close : (1 chan : Channel Close) -> L ()
fork : ((1 chan : Server p) -> L ()) -> L {use=1} (Client p)
```

First, let us see how to describe dyadic protocols such that a *client* and *server* are guaranteed to be synchronised. We describe protocols via a *global* session type:

```
data Protocol : Type -> Type where
     Request : (a : Type) -> Protocol a
     Respond : (a : Type) -> Protocol a
     (>>=) : Protocol a -> (a -> Protocol b) -> Protocol b
     Done : Protocol ()
```

A protocol involves a sequence of `Request`s from a client to a server, and `Response`s from the server back to the client. For example, we could define a protocol (Listing 5) in which a client sends a `Command` to either `Add` a pair of `Int`s or `Reverse` a `String`.

Listing 5 A global session type describing a protocol where a client can request either adding two `Int`s or reversing a `String`

```
data Command = Add | Reverse

Utils : Protocol ()
Utils = do cmd <- Request Command
           case cmd of
                Add => do Request (Int, Int)
                          Respond Int
                          Done
                Reverse => do Request String
                              Respond String
                              Done
```

`Protocol` is a DSL for describing communication patterns. Embedding it in a dependently typed host language gives us dependent session types for free, as we will see in more detail at the end of this section. We use the embedding to our advantage in a small way, by having the protocol depend on `cmd`, the command sent by the client. We can write functions to calculate the protocol for the client and the server:

```
AsClient, AsServer : Protocol a -> Actions
```

We omit the definitions, but each translates `Request` and `Response` directly to the appropriate `Send` or `Receive` action. We can see how `Utils` translates into a type for the client side by running `AsClient Utils`:

```
Send Command (\res => (ClientK
    (case res of
          Add => Request (Int, Int) >>= \_ =>
                 Respond Int >>= \_ Done
          Reverse => Request String >>= \_ =>
                     Respond String >>= \_ Done)
```

Most importantly, this shows us that the first client side operation must be to send a `Command`. The rest of the type is calculated from the command which is sent; `ClientK` is internal to `AsClient` and calculates the continuation of the type. Using these, we can define the type for `fork`.

```
Client, Server : Protocol a -> Type
Client p = Channel (AsClient p)
Server p = Channel (AsServer p)

fork : ((1 chan : Server p) -> L ()) -> L {use=1} (Client p)
```

The type of `fork` ensures that the client and the server are working to the same protocol, by calculating the channel type of each from the same global protocol. Since each channel is linear, both ends of the protocol must be run to completion.

Listing 6 An implementation of a server for the `Utils` protocol

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         case cmd of
              Add => do (x, y) # chan <- recv chan
                        chan <- send chan (x + y)
                        close chan
              Reverse => do str # chan <- recv chan
                            chan <- send chan (reverse str)
                            close chan
```

Listing 6 shows a complete implementation of a server for the `Utils` protocol. However, we do not typically write a complete implementation in one go. Idris 2's support for *holes* means that it is more convenient to write the server incrementally, in a type-driven way. We begin with just a skeleton definition, and look at the hole for the right hand side:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan = ?utilServer_rhs

 1 chan : Channel (Recv Command (\res => ... ))
--------------------------------------
utilServer_rhs : L ()
```

Therefore, the first action on `chan` must be to receive a `Command`:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         ?utilServer_rhs

   cmd : Command
 1 chan : Channel (ServerK (case cmd of ...) ...)
------------------------------
utilServer_rhs : L ()
```

We elide the full details of the type of `chan` at this stage, but at the top level it suggests that we can make progress by a `case` split on `cmd`:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         case cmd of
              Add => ?process_add
              Reverse => ?process_reverse
```

We make essential use of dependent `case` here, in that both branches have a different type which is computed from the value of the scrutinee `cmd`, similarly to `PrintfType` in Section 2.3.1. Now, for each of the holes `process_add` and `process_reverse` we see more concretely how the protocol should proceed. e.g. for `process_add`:

```
   1 chan : Channel (Recv (Int, Int) (\res =>
                    (Send Int (\res => Close))))
     cmd : Command
  -------------------------------------
  process_add : L ()
```

This shows we have to receive a pair of `Int`s, then send an `Int`. Programming is thus a *dialogue* with the type checker. Rather than trying to work out the complete program, with increasing frustration as the type checker rejects our attempts, we write the program step by step, and ask the type checker for more information on the variables in scope and the required result.

Dependent Session Types

We have the full language available at the type level, and we have already used this to our advantage in the definition of `Utils`, by using a `case` expression to choose how the protocol proceeds based on a sent value. This is a *dependent* session type, in that the protocol depends on run time information, although we have only used this to encode a form of choice. More interestingly, we can write functions which dynamically construct protocol descriptions. For example, the following function describes a server which sends back `n` values of type `a`:

```
GetN : (n : Nat) -> (a : Type) -> Protocol ()
GetN Z a = Done
GetN (S l) a = do Respond a
                  GetN l a
```

We can use this, for example, in a protocol in which a client sends a `Nat`, and the server responds with that many `String`s:

```
DepProto : Protocol ()
DepProto = do num <- Request Nat
              GetN num String
```

Listing 7 shows one possible implementation of a client for this session type, which sends a value, then receives exactly that many `String`s on the channel.

Listing 7 An implementation of a client using a dependent session type

```
depClient : (1 chan : Client DepProto) -> Nat -> L IO ()
depClient chan k
    = do chan2 <- send chan k
         getN k chan2
  where
    getN : (x : Nat) -> (1 chan : Client (GetN x String)) -> L IO ()
    getN 0 chan = close chan
    getN (S k) chan = do str # chan <- recv chan
                         putStrLn str
                         getN k chan
```

By embedding our session types implementation in a language with linear and dependent types, we need *no* extensions for dependent session types: the same machinery works for both standard and dependent sessions, thanks to the features of the host language.

Extension: Sending Channels over Channels

A useful extension is to allow a server to start up more worker processes to handle client requests. This would require sending the server's `Channel` endpoint to the worker process. However, we cannot do this with `send` as it stands, because the value sent must be of multiplicity ω, and the `Channel` is linear. One way to support this would be to refine `Protocol` to allow flagging messages as linear, then add:

```
send1 : (1 chan : Channel (Send1 ty next)) -> (1 val : ty) ->
        L {use=1} (Channel (next val))
```

This takes advantage of QTT's ability to parameterise types by linear variables like `val` here. A worker protocol, using the `Utils` protocol above, could then be described as follows, where a server forks a new worker process and immediately sends it the communication `Channel` for the client:

```
MakeWorker : Protocol ()
MakeWorker = do Request1 (Server Utils); Done
```

We leave full details of this implementation for future work. It is, nevertheless, a minor adaptation of the session types library.

6 Related Work

Substructural Types

Linear types [45] and other substructural type systems have several applications, e.g. verifying unique access to external resources [17] and as a basis for session types [21]. These applications typically use domain specific type systems, rather than the generality which would be given by

full dependent types. There are also several implementations of linear or other substructural type systems in functional languages [44, 36, 16, 33]. While these languages do not have full dependent types, Granule [36] allows many of the same properties to be expressed with a sophisticated notion of graded types which allows quantitative reasoning about resource usage, and work is in progress to add dependent types to Granule [32, 13]. ATS [40] is a functional language with linear types with support for theorem proving, which allows reasoning about resource usage and low level programming. An important mainstream example of the benefit of substructural type systems is Rust[4] [24] which guarantees memory safety of imperative programs without garbage collection or any run time overhead, and is expressive enough to implement session types [23].

Historically, combining linear types and dependent types in a fully general way – with first-class types, and the full language available at the type level – has been a difficult problem, primarily because it is not clear whether to count variable usages in types. The problem can be avoided [26] by disallowing dependent linear functions or by limiting the form of dependency [19], but these approaches limit expressivity. For example, we may still want to reason about linear variables which have been consumed. Or, as we saw at the end of Section 5, we may want to use a linear value as part of the computation of another type. Quantitative Type Theory [4, 29], allows full dependent types with no restrictions on whether variables are used in types or terms, by checking terms at a specific multiplicity.

Erasure

While linearity has benefits in allowing reasoning about effects and resource usage, one of the main motivations for using QTT is to give a clear semantics for erasure in the type system. We distinguish *erasure* from *relevance*, meaning that erased arguments are still relevant during type-checking, but erased at run time. Early approaches in Idris include the notion of "forced arguments" and "collapsible data types" [7], which give a predictable, if not fully general, method for determining which arguments can be erased. Idris 1 uses a whole program analysis [42], partly inspired by earlier work on Erasure Pure Type Systems [31] to determine which arguments can be erased, which works well in practice but doesn't allow a programmer to require specific arguments to be erased, and means that separate compilation is difficult. The problem of what to erase also exists in Haskell to some extent, even without full dependent types, when implementing zero cost coercions [46]. Our experience of the 0 multiplicity of QTT so far is that it provides the cleanest solution to the erasure problem, although we no longer infer which other arguments can be erased.

Reasoning about Effects

One of the motivations for using QTT beyond expressing erasure in types is that it provides a core language which allows reasoning about external resource usage. Previous work on reasoning about effects and resources with dependent types has relied on indexed monads [3, 27] or embedded DSLs for describing effects [9]. These are effective, but generally difficult to compose; even if we can compose effects in a single EDSL, it is hard to compose multiple EDSLs, especially when parameterised with type information. Other successful approaches such as Hoare Type Theory [34] are sufficiently expressive, but difficult to apply in everyday programming. Having linear types in the core language means that tracking state changes, which we have previously had to encode in a state-tracking monad, is now possible directly in the language. We can compose multiple resources by using multiple linear arguments.

[4] `https://rust-lang.org/`

Combining dependent and linear types, along with protocol descriptions in `L`, gives us similar power to Typestate [1, 48], in that we can use dependency to capture the state of a value in its type, and linearity to ensure that it is always used in a valid state. First-class types gives us additional flexibility: we can reason about state changes which are only known at run-time, such as checking a PIN in an ATM.

Session Types

In Section 5 we gave an example of using QTT to implement Dyadic Session Types [21]. In previous work [11] Idris has been experimentally extended with uniqueness types, to support verification of concurrent protocols. However, this earlier system did not support erasure, and as implemented it was hard to combine unique and non-unique references. Our experience with QTT is that its approach to linearity, with multiplicities on the binders rather than on the types, is much easier to combine with other non-linear programs.

Given linearity and dependent types, we can already have dependent session types, where, for example, the progress of a session depends on a message sent earlier. Thus, the embedding gives us label-dependent session types [43] with no additional cost. Previous work in exploring value-dependent sessions in a dependently typed language [15] is directly expressible using linearity in Idris 2. We have not yet explored further extensions to session types, however, such as multiparty session types [22], dealing with exceptions during protocol execution [18] or dealing with errors in transmission in distributed systems.

7 Conclusions and Further Work

Implementing Idris 2 with Quantitative Type Theory in the core has immediately given us a lot more expressivity in types than Idris 1. For most day to day programming tasks, expressing erasure at the type level is the most valuable user-visible new feature enabled by QTT, in that it is unambiguous which function and data arguments will be erased at run time. Erasure has been a difficulty for dependently typed languages for decades and until recently has been handled in partial and unsatisfying ways (e.g. [12]). Quantitative Types, and related recent work [42], are the most satisfying so far, in that they give the programmer complete control over what is erased at run time. In future, we may consider combining QTT with inference for additional erasure [42].

The 1 multiplicity enables programming with full linear dependent types. Therefore reasoning about resources, which previously required heavyweight library implementations, is now possible directly, in pure functions. We have also seen, briefly, that quantities give more information when inspecting the types of holes. More expressive types, with interactive editing tools. make programming a *dialogue* with the machine, rather than an exercise in frustration when submitting complete (but wrong!) programs to the type checker.

We have often found full dependent types, where a type is a first class language construct, to be extremely valuable in developing libraries with expressive interfaces, even if the programs which use those libraries do not use dependent types much. The `L` type for embedding linear protocols is an example of this, in that it allows a programmer to express precisely not only *what* a function does, but also *when* it is allowed to do it. It is important that the type system remains accessible to programmers, however. Dependent and linear types are powerful concepts, and without care in library design, can be hard to use. However, they don't have to be: they are based on concepts that programmers routinely understand and use, such as using a variable once and making assumptions about the relationships between data. A challenge for language and tool designers is to find the right syntax and feedback mechanisms, so that powerful verification tools are within reach of all software developers.

While we have already found many benefits of being able to express quantities in types, we have only just begun exploring, and have encountered some limitations in the theory which we hope to address, perhaps adapting ideas from related work [13]. Most importantly, we would like to express *polymorphic* quantities. This may, for example, help give an appropriate type to >>= taking into account that some monads guarantee to execute the continuation exactly once, but others need more flexibility. Similarly, like Granule [36], we may find it useful to use quantities other than `0` and `1`, and the theory behind QTT supports this.

We have not discussed performance in this paper, but for an interactive system it is vital, and will be a primary concern in the near future. Following [25], Idris 2 minimises substitution of unification solutions. Initial results are promising: Idris 2 is now self-hosting, and builds itself in around 90 seconds[5]. We are using the interactive development tools, especially holes, in developing Idris itself.

Finally, an important application of reasoning about linear resource usage is in implementing communication and security protocols correctly. The `Protocol` type in Section 5 provides a preliminary example which demonstrates the possibilities, but realistically it will need to handle timeouts, exceptions and more sophisticated protocols. Implementing these protocols correctly is difficult and error prone, and errors lead to damaging security problems[6]. But in describing a session type, we have explained a protocol in detail, and the machine calculates a lot of information about how the protocol proceeds. We should not let the type checker keep this information to itself! Thus, interactive programming of protocols based on linear resource usage gives a foundation for secure programming.

References

1 J Aldrich, J Sunshine, D Saini, and Z Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1012, 2009. URL: `http://dl.acm.org/citation.cfm?id=1640073`.

2 Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-Scope Safe Programs and their Proofs. In *CPP*, pages 195–207, 2017.

3 Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, 2009. `doi:10.1017/S095679680900728X`.

4 Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS 2018*, 2018. `doi:10.1145/3209108.3209189`.

5 L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*. Citeseer, 1999. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2895`.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, December 2017. `doi:10.1145/3158093`.

7 Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

8 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013.

9 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming (TFP '14)*, volume 8843 of *LNCS*. Springer, 2014.

[5] Dell XPS 13 Laptop, running Ubuntu 18.03 LTS

[6] e.g. `https://www.imperialviolet.org/2014/02/22/applebug.html`

10 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.

11 Edwin Brady. Type-driven development of concurrent communicating systems. *Computer Science*, 18(3), 2017.

12 Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

13 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie C. Weirich. A graded dependent type system with a usage-aware semantics (extended version). In *arXiv:2011.04070 [cs]*, January 2021. arXiv: 2011.04070. URL: http://arxiv.org/abs/2011.04070.

14 Nils Anders Danielsson. Total parser combinators. In *International Conference on Functional Programming (ICFP 2010)*, 2010. doi:10.1145/1932681.1863585.

15 Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. Value-dependent session design in a dependently typed language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 47–59, 2019. doi:10.4204/EPTCS.291.5.

16 Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages*, pages 201—-218, 2008.

17 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In David Schmidt, editor, *Programming Languages and Systems*, pages 204–218, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

18 Simon Fowler, Sam Lindley, J Garrett Morris, and Sara Decova. Exceptional Asynchronous Session Types: Session Types without Tiers. In *Principles of Programming Languages (POPL 2019)*, 2019.

19 Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 357, 2013. doi:10.1145/2429069.2429113.

20 Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD Thesis, University of Strathclyde, 2013. URL: https://personal.cis.strath.ac.uk/adam.gundry/thesis/thesis-2013-07-24.pdf.

21 Kohei Honda. Types for dyadic interaction. In *CONCUR 1993 (International Conference on Concurrency Theory)*. Springer, 1993.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages (POPL 2008)*, 2008.

23 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session Types for Rust. In *WGP 2015 (Workshop on Generic Programming)*. ACM, 2015.

24 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158154.

25 András Kovács. Fast elaboration for dependent type theories, 2019. Talk at EU Types WG Meeting.

26 Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Dependent and Linear Types. In *Principles of Programming Languages (POPL 2015)*, 2015.

27 Conor McBride. Kleisli arrows of outrageous fortune, 2011.

28 Conor McBride. How to Keep Your Neighbours in Order. In *International Conference on Functional Programming (ICFP 2014)*, 2014.

29 Conor McBride. I got plenty o' nuttin'. In *A List of Successes that Can Change the World*, 2016.

30 Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992. URL: http://www.sciencedirect.com/science/article/pii/074771719290011R.

31 Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008. doi:10.1007/978-3-540-78499-9_25.

32 Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded Modal Dependent Type Theory. In *ESOP 2021*, 2020. arXiv: 2010.13163. URL: http://arxiv.org/abs/2010.13163.

33 J Garrett Morris. The Best of Both Worlds: Linear Functional Programming Without Compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*, pages 448–461, 2016. doi:10.1145/2951913.2951925.

34 Aleksander Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP 2008)*, pages 229—-240, 2008. doi:10.1145/1411204.1411237.

35 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7934&rep=rep1&type=pdf.

36 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), 2019. doi:10.1145/3341714.

37 Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School*, pages 47—-96, 2001.

38 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-lezama. Program Synthesis from Polymorphic Refinement Types. *PLDI*, 2016. ISBN: 9781450342612.

39 A. H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. doi:10.1109/PROC.1967.5493.

40 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Science of Computer Programming*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.

41 Matúš Tejiščák. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.*, 4(ICFP):91:1–91:29, 2020. doi:10.1145/3408973.

42 Matúš Tejiščák. *Erasure in Dependently Typed Programming*. PhD thesis, University of St Andrews, 2020.

43 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371135.

44 Jesse a. Tov and Riccardo Pucella. Practical affine types. In *Principles of Programming Languages*, pages 447—-458, 2011. doi:10.1145/1925844.1926436.

45 Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

46 Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. A role for dependent types in Haskell. *Proc. ACM Program. Lang.*, 3(ICFP):101:1–101:29, 2019. doi:10.1145/3341705.

47 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, 2017. doi:10.1145/3110275.

48 Roger Wolff, Jonathan Aldrich, Ronald Garcia, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *Transactions on Programming Languages and Systems*, 36(4):1–44, 2014.