Idris 2: Quantitative Type Theory in Action

EDWIN BRADY, University of St Andrews, Scotland, UK

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 several examples of the benefits of QTT in practice including: clearly expressing which data is erased at run time, at the type level; improving interactive program development by reducing the search space for type-driven program synthesis; and, resource tracking in the type system leading to type-safe concurrent programming with session types.

1 INTRODUCTION

Dependently typed programming languages, such as Idris [Brady 2013], Agda [Norell 2007], and Haskell with the appropriate extensions enabled [Weirich et al. 2017], 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 [McBride 2014], termination of parsing [Danielsson 2010] and scope safety of programs [Allais et al. 2017]. 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 [Brady 2014] or indexed monads [Atkey 2009; McBride 2011]. These are, nevertheless, heavyweight techniques which can be hard to compose.

Substructural type systems, such as linear type systems [Bernardy et al. 2017; Morris 2016; Wadler 1990], 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 [Krishnaswami et al. 2015] by never allowing types to depend on a linear term, but more recent work on Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016] solves the problem by assigning a *quantity* to each binder, and checking terms at a specific *multiplicity*. Briefly, in QTT a variable has a multiplicity: 0, 1 or unrestricted (ω). We can freely use any variable at multiplicity 0—e.g., in types—but we can not use a variable with multiplicity 0 at any other multiplicity, and a variable 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.

Author's address: Edwin Brady, School of Computer Science, University of St Andrews, Jack Cole Building, St Andrews, KY16 9SX, Scotland, UK, ecb10@st-andrews.ac.uk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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 unification, interfaces, `do`-notation, dependent `case` expressions and other syntactic sugar. We discuss how the features of QTT affect the high level language design. We also consider how to structure larger applications, and how to program with convenient library features such as exceptions while still supporting linearity where necessary. As an example, we will show how to implement a library for concurrent programming with session types [Honda 1993]. The code is submitted as anonymised supplementary material (`idris2-code.tgz`).

1.1 Contributions

This paper is about exploring what is possible in a language based on Quantitative Type Theory (QTT). We make the following research contributions:

- We introduce Idris 2 (Section 2), a new version of Idris based on QTT, where each binder is associated with a quantity. We describe how QTT has influenced the language design by allowing type-level support for *erasure* and *linearity*, and show how quantities on variables help type-driven interactive editing.
- We demonstrate how the combination of linear and dependent types allows the implementation and verification of resource usage protocols (Section 3), illustrating with a data store which statically requires successful login before accessing data.
- We describe how to structure larger Idris applications (Section 4), using a type `App` which allows us to describe the interactive actions, states and exceptions that a function supports, and illustrate with a detailed example, dependent session types (Section 5).

Importantly, `App` allows us to distinguish *linear* program fragments—those which execute exactly once, and are guaranteed to return a result—from those which may throw exceptions. In this way, we are able to safely include linear resource protocols as components of larger systems, knowing that protocols which must run to completion actually do so.

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 investigating the new kinds of programming and reasoning it enables. Where appropriate, we will discuss the intuition behind understanding how argument multiplicities work in practice.

2 IDRIS 2 AND QUANTITATIVE TYPES

Idris 2 is a new version of Idris, implemented in Idris 1, and based on Quantitative Type Theory (QTT) as defined in recent work by Atkey [Atkey 2018] following initial ideas by McBride [McBride 2016]. 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 in detail shortly. Several factors have motivated the new implementation:

- In implementing Idris in itself, we will necessarily do 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 [Brady 2005; Tejiscak 2020]. 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, it has outgrown the requirements of its initial experimental implementation, and since significant re-engineering has been required, now is a good time to start a re-implementation in Idris itself.

The new core language has led to several small changes in the surface language[1], particularly with regard to run time erasure. In this section, we will discuss these changes, and give an overview of quantitative types and their applications in general.

2.1 Run Time Erasure in Types

Consider the following skeleton function definition (deliberately chosen for its familiarity!):

```
append : Vect n a -> Vect m a -> Vect (plus n m) a
append xs ys = ?append_rhs
```

The names `n`, `a` and `m` are *implicit* arguments to `append`, and we refer to them as *unbound implicits*. Like Idris 1, Idris 2 implicitly binds names in a type declaration which begin with a lower case letter, and appear in argument position. The `?append_rhs` on the right hand side is a *hole*, where a hole stands for a part of a function yet to be written. Idris 2 has a REPL, at which we can inspect the type of the hole and its context:

```
Main> :t append_rhs
 0 m : Nat
 0 a : Type
 0 n : Nat
   ys : Vect m a
   xs : Vect n a
-------------------------------------
append_rhs : Vect (plus n m) a
```

This explicitly shows that we have the lengths `m` and `n` of the two vectors, the element type `a` and the vectors `xs` and `ys` available to use. However, `m`, `n` and `a` are annotated with a `0`. This means we can only refer to them in an *erased* context, e.g. in a type, or in another argument position with multiplicity `0`. Arguments with multiplicity `0` are *erased*—that is, used 0 times—at run time. For example, this means we cannot write a function such as the following, which attempts to create a vector of `n` copies of a value of type `a`, by matching on the implicit argument, because the implicit argument will not be available at run time:

```
rep : a -> Vect n a
rep {n = Z} val = []
rep {n = S k} val = val :: rep val
```

This results in the error "`Can't match on Z (Erased argument)`". For this definition to be accepted, we have to change the type to state that `n` is used by the definition:

```
rep : {n : Nat} -> a -> Vect n a
```

In general, we can write function argument types in one of the following forms:

- `{x : T} -> ...` for an implicit argument which is usable with no restrictions at run time.
- `{0 x : T} -> ...` for an implicit argument which will be *erased* at run time.

[1]For this reason, Idris 2 is not yet self-hosting, but it is planned in the near future

- `{1 x : T} -> ...` for an implicit argument which will be used *exactly once* at run time. We will discuss the "use once" multiplicity in the next section.
- Similarly, `(x : T)`, `(0 x : T)`, `(1 x : T)` for giving quantities for explicit arguments.

As a syntactic shorthand, unbound implicits such as `n`, `a` and `m` in `append` are given multiplicity `0`. Alternatively, we can write...

```
append : forall n, a, m . Vect n a -> Vect m a -> Vect (plus n m) a
```

...where `forall` binds an implicit argument with multiplicity `0`. The multiplicity `0` makes it clear, in a function's type, which arguments are erased at run time.

Remark: We have not discussed details of the issue here, but erasure does not imply *irrelevance*. Erased arguments are nevertheless relevant during type checking [Tejiscak 2020].

2.2 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 [Bernardy et al. 2017], 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. To illustrate, we can try (and fail!) to write a function which duplicates a value declared as "use once", interactively:

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

Inspecting the `dup_rhs` hole shows that we have:

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

```
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`".

2.3 Auto implicit arguments

As well as implicit arguments, which are resolved by unification, Idris supports `auto`-implicit arguments, which are resolved by searching for a unique expression of the appropriate type, using data constructors and local variables as search hints, as well as explicitly delared hints. For example, we can write a *total* `fromMaybe` function as follows:

```
data IsJust : Maybe a -> Type where
     ItIsJust : IsJust (Just val)

fromMaybe : (x : Maybe a) -> {auto p : IsJust x} -> a
fromMaybe (Just x) {p = ItIsJust} = x
```

The notation {`auto x : T`} `-> ...` declares an `auto`-implicit argument, which can be annotated with multiplicities like other implicit and explicit arguments. When we apply `fromMaybe` to an argument, the type checker will try to find an appropriate implementation of `IsJust`. This will succeed if the value is of the form `Just val`, and fail otherwise:

```
Main> fromMaybe (Just 10)
10
Main> fromMaybe Nothing
(interactive):1:1--1:18:Can't find an implementation for
IsJust Nothing
```

We can use auto-implicits to implement type classes. In Idris terminology, these are called "interfaces", since there can be multiple implementations and they can be parameterised on *anything*, not only types. Given an interface declaration such as...

```
interface Show a where
  show : a -> String
```

...Idris 2 generates a data declaration and top level functions for each of the methods (just `show` here), where the name `MkShow` is automatically generated, and fresh:

```
data Show : (a : Type) -> Type where [noHints]
     MkShow : (show : forall a . a -> String) -> Show a

show : {auto con : Show a} -> a -> String
show {con = MkShow show_meth} x = show_meth x
```

The `[noHints]` annotation means that Idris will not generate a search hint for the constructor `MkShow`, because we only want the `auto`-implicit search to search explicitly written implementations of the interface. An implementation is written as follows:

```
Show Bool where
  show False = "False"
  show True = "True"
```

This translates into a top level function declaration, with a `%hint` annotation which tells the type checker to use `ShowBool` as a hint in the `auto`-implicit search:

```
%hint ShowBool : Show Bool
ShowBool = MkShow show_meth where
  show_meth False = "False"
  show_meth True = "True"
```

The notation `Show a => ...` is syntactic sugar for {`auto _ : Show a`} `-> ...`:

```
fromMaybe : (x : Maybe a) -> IsJust x => a
show : Show a => a -> String
```

Like Haskell, and like Idris 1, interfaces and implementations can themselves be constrained by `auto`-implicits, e.g.:

```
interface Eq a => Ord a where
  compare : a -> a -> Ordering
  ...
```

The `auto`-implicit implementation in Idris 2 is a significant internal difference from Idris 1, in that it consolidates `auto`-implicit search and implementation search into the same mechanism, with the same notation.

2.4 Type-driven Program Synthesis

There are several potential benefits to expressing linear argument usage, including optimisations (reducing the need for allocation and garbage collection, since values can safely be overwritten) and tracking resource usage (by making sure a value in a specific state can only be used once). Another, perhaps less immediately evident, benefit is that it can restrict the search space of type-driven program synthesis.

Idris 2 provides a (fairly unsophisticated, for the moment) type-driven program search command for use in editor interaction, implemented as a brute force search of the space of possible programs which returns the first well-typed result. A sketch of the algorithm is:

(1) Generate a skeleton definition from the type, of the form `f x1 x2 ... xn = ?f_rhs`
(2) Search for a value which fits `f_rhs` by trying all of the local variables, the constructors for the return type of `f_rhs`, recursive calls to `f` with decreasing arguments, then recursively searching for arguments on success.
(3) If step (2) fails, case split on each of `x1` to `xn` in turn, then recursively search for a valid right hand side for each of the resulting clauses.

This is similar to `auto`-implicit search, except that it takes the first well-typed result, rather than checking that the result is unique. By giving a precise enough type, it can find the intended implementation. For the previous `append` example, it finds:

```
append : Vect n a -> Vect m a -> Vect (plus n m) a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

This works because we have given, at the type level, additional information that the length of the output must be the sum of the lengths of the inputs. But examples with `Vect`, while useful for explanation and demonstration, are not always realistic. Often, in practice, a `Vect` is too constraining and a `List` will suffice, but if we try program synthesis for appending lists, the first result it finds is:

```
append : List a -> List a -> List a
append xs ys = xs
```

This is well-typed, but not what we intended! Linearity annotations give us another way to constrain the search. Here, we want to state that *both* lists `xs` and `ys` must appear in the result. We can do this with a multiplicity `1` on each of the arguments:

```
append : (1 xs : List a) -> (1 ys : List a) -> List a
```

This rules out the previous result, because it did not consume `ys`. Searching now gives:

```
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

Note that there are other well-typed results for this (e.g. swapping the order of `xs` and `ys` in the recursive call) so a programmer still needs to check the result is the intended function.

Remark: The typing rules of QTT don't require that the arguments we pass to `append` are linear, merely that they are not erased. It is valid to use an argument with multiplicity ω in a position with multiplicity 1. A linear argument is a promise that the function will use the argument exactly once, not a requirement that the argument is not used elsewhere.

2.5 I/O in Idris 2

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

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

The `%World` argument is consumed *exactly once*, so it is not possible to refer to previous world states (after all, you can't unread a file, or unplay a sound!). It returns an `IORes`:

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

This is a pair of the function's result (with unrestricted usage), and an updated world state. The intuition for multiplicities in data constructors is the same as for those in functions: here, if `MkIORes x w` is evaluated exactly once, then the corresponding world `w` is evaluated exactly once. We can wrap `PrimIO` to get `IO`:

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

There is a primitive `io_bind` operator (from which we can build a `Monad` instance), 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. It can be informative to insert a hole to see how the multiplicities are updated:

```
io_bind (MkIO fn)
    = \k => MkIO (\w => let MkIORes x' w' = fn w in ?io_bind_rhs)
```

This shows that, at the point `io_bind_rhs` is used, we have consumed `fn` and `w`, and we still have to run the continuation `k` exactly once, and use the updated world `w'` exactly once:

```
 0 b : Type
 0 a : Type
 0 fn : (1 x : %World) -> IORes a
 1 k : a -> IO b
 0 w : %World
```

```
1 w' : %World
  x' : a
-------------------------------------
io_bind_rhs : IORes b
```

This implementation of `IO` is similar to the approach taken in Haskell [Peyton Jones 2001], 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. While precise, this can be unwieldy in practice since most I/O operations might fail, so we will revisit this in Section 4.

3 RESOURCE USAGE PROTOCOLS

The `IO` implementation, via a linearly consumed `%World` token, illustrates how we can use quantitative types to ensure that there is a unique reference to an external resource. There is only one `%World`, so it would not make sense to duplicate it, delete it, or try to access previous versions of it. Taking inspiration from Clean[2] which uses unique references to external resources such as files, we can do something similar for other resources and use dependent types to track the abstract state of a resource in the process.

3.1 A Password Protected Data Store

Consider an online data store, holding some secret data, which we can only access after logging in. The store has two states: `LoggedIn`, which means that a user is logged in and able to access data; and `LoggedOut`, which means that the user is not logged in. Reading data is only allowed when the store is in the `LoggedIn` state. Figure 1 illustrates the states, operations, and transitions on the data store.

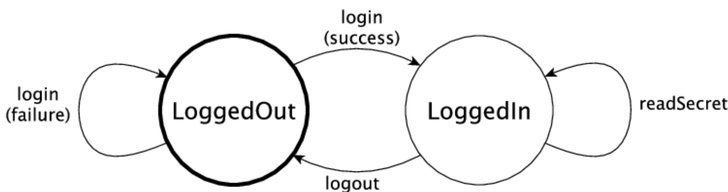

Fig. 1. A state machine describing the states and transitions in a system which allows a program to read some secret data, only after successfully logging in

If we represent the store using a linear type, then as with `%World`, we can ensure that there is a unique reference to a store, and we can control which operations are allowed at which time. Listing 1 shows the interface to the data store in full. We will elaborate in the following sections.

[2] https://wiki.clean.cs.ru.nl/Clean

Listing 1. Linear Interface to a Data Store

```
data Res : (a : Type) -> (a -> Type) -> Type where
     (@@) : (val : a) -> (1 resource : r val) -> Res a r

data Access = LoggedOut | LoggedIn
data Store : Access -> Type

connect : (1 k : (1 s : Store LoggedOut) -> IO a) -> IO a
disconnect : (1 s : Store LoggedOut) -> IO ()

login : (1 s : Store LoggedOut) -> (password : String) ->
        Res Bool (\ok => Store (if ok then LoggedIn else LoggedOut))
logout : (1 s : Store LoggedIn) -> Store LoggedOut
readSecret : (1 s : Store LoggedIn) ->
             Res String (const (Store LoggedIn))
```

3.2 Defining the Store

We define a type `Access` for the possible states of a store, then parameterise a `Store` by whether it is currently logged in or logged out:

```
data Access = LoggedOut | LoggedIn
data Store : Access -> Type
```

We leave the details of the `Store` abstract. In practice it might be a file, database handle, or some other reference to external data. What is important is that the reference is *linear*: a reference to a store can only be accessed once, so if an operation changes the state of a store, the old and no longer valid store can not be accessed.

In QTT, multiplicities are associated with *binding occurrences*, rather than types, so we can not state that all data stores are used linearly. Instead, we create them via a *continuation*:

```
connect : (1 k : (1 s : Store LoggedOut) -> IO a) -> IO a
```

This is the only way we provide in the interface to create a new reference to a data store, so any `Store` we have in a program will be linear. Since it is linear it must eventually be consumed. We do so by disconnecting, provided that it is in the `LoggedOut` state:

```
disconnect : (1 s : Store LoggedOut) -> IO ()
```

While we have a `Store` available, we can run the operations `login`, `logout` and `readSecret` as illustrated in Figure 1. These operations take a linear data store as input, and return an updated data store as output, possibly with some additional data. If we want to return a result along with the updated store, we use the following type `Res`:

```
data Res : (a : Type) -> (a -> Type) -> Type where
      (@@) : (val : a) -> (1 resource : r val) -> Res a r
```

This is a *dependent pair* of an unrestricted value `val`, and a resource, the type of which is computed from `val`. In the same way as `IORes`, because the resource argument is marked as linear, consuming a value of the form `val @@ res` exactly once means that the value `res` is consumed exactly once. The `login` operation, for example, takes a password and returns a pair of whether logging in succeeded, and a data store in the appropriate state:

```
login : (1 s : Store LoggedOut) -> (password : String) ->
        Res Bool (\ok => Store (if ok then LoggedIn else LoggedOut))
```

We also provide a `logout` operation, which returns an updated store in the `LoggedOut` state, and a `readSecret` operation which will only read from a `LoggedIn` store. Since the store is linear, and consumed, it returns a new reference to the logged in store, no matter what the result string (hence the use of `const : a -> b -> a`):

```
logout : (1 s : Store LoggedIn) -> Store LoggedOut
readSecret : (1 s : Store LoggedIn) ->
             Res String (const (Store LoggedIn))
```

We can illustrate how this works in practice, and how the resource type changes throughout a program's execution, by writing a program interactively which logs in, reads the secret data if logging in succeeded, closes the store, and returns the secret on success.

3.3 Accessing the Store: Valid Protocol Usage

We can write a program that logs in to a store, reads the secret if successful, then logs out, interactively, using holes to see how the state of the store changes with each operation. We begin by trying to login with a hard coded password:

```
storeProg : IO (Maybe String)
storeProg = connect $ \store =>
    let ok @@ store = login store "Mornington Crescent" in
        ?what_next
```

We can inspect the hole `what_next` to see the updated state of the data store:

```
   ok : Bool
 1 store : Store (if ok then LoggedIn else LoggedOut)
-------------------------------------
what_next : IO (Maybe String)
```

Logging in returns a result `ok` which tells us whether it succeeded. But, as the type of `store` suggests, we can only know the updated state of the store by inspecting `ok`:

```
storeProg : IO (Maybe String)
storeProg = connect $ \store =>
    let ok @@ store = login store "Mornington Crescent" in
        if ok then ?success else ?failure
```

Now the type of `store` in the context of the holes `success` and `failure` has been refined according to the value of `ok` in each branch. For example, in `success`;

```
 1 store : Store LoggedIn
   ok : Bool
-------------------------------------
success : IO (Maybe String)
```

Remark: The `if...then...else` expression elaborates to core QTT via a dependent `case` operator, where each branch of the case has a different type, depending on the value of the scrutinee. Even this small example shows the value of a dependent `case` operator for effective programming of protocols. The states involved (and hence the types) are different in each branch of the `case`, reflecting different states in different exections of the protocol. We find that a dependent `case` is essential!

Once successfully logged in, we can either read the secret or log out. In each case, the type of the store being returned explicitly shows the result of the corresponding state transition:

```
logout : (1 s : Store LoggedIn) -> Store LoggedOut
readSecret : (1 s : Store LoggedIn) ->
                Res String (const (Store LoggedIn))
```

Listing 2 shows the completed `storeProg` program, rebinding `store` at each point as its state updates. The linear type of `store` throughout ensures that only operations which are valid at that point can be executed. It is impossible to read the secret if logging in failed; this would lead to an error of the form "`Mismatch between: LoggedOut and LoggedIn`".

Listing 2. A function which follows the Data Store protocol, to read a string if the password is correct

```
storeProg : IO (Maybe String)
storeProg = connect $ \store =>
    let ok @@ store = login store "Mornington Crescent" in
        if ok then let secret @@ store = readSecret store
                       store = logout store in
                       do disconnect store
                          pure (Just secret)
              else do disconnect store
                      pure Nothing
```

do-notation and monads. In Idris, `do`-notation translates syntactically to applications of `(>>=)`, before type checking. There is a `(>>=)` defined in the `Monad` interface, partially declared as follows (it also includes `join`):

```
Applicative m => Monad m where
  (>>=) : m a -> (a -> m b) -> m b
```

There are no linearity annotations here: the continuation can be run as many times as we like, which is important for many monads including `Maybe` and `List`. We don't yet have polymorphism over quantities in QTT (unlike Linear Haskell) and, in any case, as an experimental new language feature, we believe any changes to the interface declarations in the Prelude should be left to future versions for ease of transition to Idris 2.

Nevertheless, it is a problem here. We need to be sure that a linear variable is used exactly once, and the continuation of `(>>=)` may be run multiple times so can't use any linear variable. To solve this we define our own `(>>=)` locally to overload `do`-notation for `IO` specifically, implemented using the `io_bind` primitive seen in Section 2.5:

```
(>>=) : (1 act : IO a) -> (1 k : a -> IO b) -> IO b
(>>=) = io_bind
```

Idris resolves name ambiguities by type, and takes a pragmatic approach (which we may revisit in the future) to resolving ambiguities such as this by choosing the name with a concrete return type (so prioritising `IO b` over `m b`.)

4 INTERACTIVE APPLICATIONS IN IDRIS 2

Idris applications have `main : IO ()` as an entry point, and we have seen how `IO` is defined internally using a linear reference to a primitive `%World`. This is fine for primitives, but `IO` does not support exceptions so we have to be explicit about how an operation handles failure.

Also, if we do extend it to support exceptions, we will not be able to use linear protocols as described in Section 3, because throwing an exception would mean that the protocol is never completed, violating linearity.

In this section, we describe a parameterised type `App`, and a related parameterised type `App1`, which together allow us to structure larger applications, taking into account both exceptions and linearity. The aims of `App` and `App1` are that they should:

- make it possible to express what interactions a function does, in its type, without too much notational overhead.
- have little or no performance overhead compared to writing in `IO`.
- be compatible with other libraries and techniques for describing effects, such as algebraic effects or monad transformers.
- be sufficiently easy to use and performant that it can be the basis of *all* libraries that make foreign function calls, much as `IO` is in Idris 1 and Haskell.
- most importantly for the present paper, be compatible with linear types, meaning that they should express whether a section of code is linear (guaranteed to execute exactly once without throwing an exception) or whether it might throw an exception.

We begin by introducing `App`, with some example programs, then show how to extend it with exceptions, state, and other interfaces. Finally, we show how the design allows it to safely interoperate with linear resources, revisiting the data store from Section 3.

4.1 Introducing App

`App` is declared as below, in a module `Control.App`. It is parameterised by an implicit `Path` (which states whether the program's execution path is linear or might throw exceptions), which has a `default` value that the program might throw, and an `Environment` (which gives a list of exception types which can be thrown, and is a synonym for `List Type`):

```
data App : {default MayThrow l : Path} ->
           (e : Environment) -> Type -> Type
```

It serves the same purpose as `IO`, and is implemented similarly via a reference to `%World`, but is more informative. To use `App` in general, we typically constrain `e` by the interfaces it supports. e.g. a program which supports console IO:

```
hello : Console e => App e ()
hello = putStrLn "Hello, App world!"
```

Or, a program which supports console IO and carries an `Int` state, labelled `Counter`:

```
helloCount : (Console e, State Counter Int e) => App e ()
helloCount = do c <- get Counter
                put Counter (c + 1)
                putStrLn "Hello, counting world"
```

For convenience, we can list multiple interfaces in one go, using a function `Has` to compute the interface constraints:

```
helloCount : Has [Console, State Counter Int] e => App e ()

0 Has : List (a -> Type) -> a -> Type
Has [] es = ()
Has (e :: es') es = (e es, Has es' es)
```

The purpose of `Path` is to state whether a program can throw exceptions, so that we can know where it is safe to reference linear resources. It is declared as follows:

```
data Path = MayThrow | NoThrow
```

The type of `App` states that `MayThrow` is the default. We expect this to be the most common case. After all, realistically, most operations have possible failure modes, especially those which interact with the outside world. The `0` on the declaration of `Has` indicates that it can only be run in an erased context, so it will never be run at run-time. To run an `App` inside `IO`, we use an initial environment `Init` (recall that an `Environment` is a `List Type`):

```
Init : Environment
Init = [Void]

run : App {l} Init a -> IO a
```

Generalising the `Path` parameter with `l` means that we can invoke `run` for any application, whether the `Path` is `NoThrow` or `MayThrow`. But, in practice, all applications given to `run` will not throw at the top level, because the only exception type available is the empty type `Void`. Any exceptions will have been introduced and handled inside the `App`.

4.2 Exceptions

The `Environment` is a list of error types, usable via the `Exception` interface:

```
interface Exception err e where
  throw : err -> App e a
  catch : App e a -> (err -> App e a) -> App e a
```

We can use `throw` and `catch` for some exception type `err` as long as the exception type exists in the environment:

```
data HasErr : Type -> Environment -> Type where
     Here : HasErr e (e :: es)
     There : HasErr e es -> HasErr e (e' :: es)

HasErr err e => Exception err e where ...
```

Note the `HasErr` constraint on `Exception`: this is one place where it is notationally convenient that the `auto`-implicit mechanism and the interface resolution mechanism are identical. Finally, we can introduce new exception types via `handle`, which runs a block of code which might throw, handling any exceptions:

```
handle : App (err :: e) a -> (onok : a -> App e b) ->
         (onerr : err -> App e b) -> App e b
```

4.3 Adding State

Applications will typically need to keep track of state, and we support this primitively in `App` using a `State` type:

```
data State : (tag : a) -> Type -> Environment -> Type
```

The `tag` is used purely to distinguish between different states, and is not required at run-time, as explicitly stated in the types of `get` and `put`:

```
get : (0 tag : a) -> State tag t e => App {l} e t
put : (0 tag : a) -> State tag t e => t -> App {l} e ()
```

These use an `auto`-implicit to pass around a `State` with the relevant `tag` implicitly, so we refer to states by tag alone. In `helloCount`, we used an empty type `Counter` as the tag:

```
data Counter : Type where -- complete definition
```

The environment `e` is used to ensure that states are only usable in the environment in which they are introduced. They are introduced using `new`:

```
new : t -> (1 p : State tag t e => App {l} e a) -> App {l} e a
```

Note that the type tells us `new` runs the program with the state exactly once. Rather than using `State` and `Exception` directly, however, we typically use interfaces to constrain the operations which are allowed in an environment. Internally, `State` is implemented via an `IORef`, primarily for performance reasons.

4.4 Defining Interfaces for App

The only way provided by `Control.App` to run an `App` is via the `run` function, which takes a concrete environment `Init`. All concrete extensions to this environment are via either `handle`, to introduce a new exception, or `new`, to introduce a new state. In order to compose `App` programs effectively, rather than introducing concrete exceptions and state in general, we define interfaces for collections of operations which work in a specific environment.

4.4.1 Example: Console I/O. We have seen an initial example using the `Console` interface, which is declared as follows:

```
interface Console e where
  putStr : String -> App {l} e ()
  getStr : App {l} e String
```

It provides primitives for writing to and reading from the console, and generalising the path parameter to `l` means that neither can throw an exception, because they have to work in both the `NoThrow` and `MayThrow` contexts.

To implement this for use in a top level `IO` program, we need access to primitive `IO` operations. The `Control.App` library defines a primitive interface for this:

```
interface PrimIO e where
  primIO : IO a -> App {l} e a
  fork : (forall e' . PrimIO e' => App {l} e' ()) -> App e ()
```

We use `primIO` to invoke an `IO` function. We also have a `fork` primitive, which starts a new thread in a new environment supporting `PrimIO`. Note that `fork` starts a new environment `e'` so that states are only available in a single thread.

There is an implementation of `PrimIO` for an environment which can throw the empty type as an exception. This means that if `PrimIO` is the only interface available, we cannot throw an exception, which is consistent with the definition of `IO`. This also allows us to use `PrimIO` in the initial environment `Init`.

```
HasErr Void e => PrimIO e where ...
```

Given this, we can implement `Console` and run our `hello` program in `IO`:

```
PrimIO e => Console e where
  putStr str = primIO $ putStr str
  getStr = primIO $ getLine

main : IO ()
```

```
main = run hello
```

Or, by initialising the state too, we can run our previous `helloCount` program:

```
mainCount : IO ()
mainCount = run (new 93 helloCount)
```

4.4.2 Example: File I/O. Console I/O can be implemented directly, but most I/O operations can fail. For example, opening a file can fail for several reasons: the file does not exist; the user has the wrong permissions, etc. In Idris, the `IO` primitive reflects this in its type:

```
openFile : String -> Mode -> IO (Either FileError File)
```

While precise, this becomes unwieldy when there are long sequences of `IO` operations. Using `App`, we can provide an interface which throws an exception when an operation fails, and guarantee that any exceptions are handled at the top level using `handle`. We begin by defining the `FileIO` interface:

```
interface Has [Exception IOError] e => FileIO e where
  withFile : String -> Mode ->  (onError : IOError -> App e a) ->
             (onOpen : File -> App e a) -> App e a
  fGetStr : File -> App e String
  fPutStr : File -> String -> App e ()
  fEOF : File -> App e Bool
```

We use resource bracketing, rather than an explicit `open` operation, to open a file, to ensure that the file handle is cleaned up on completion[3]. All of the operations can fail, and the interface makes this explicit by saying we can only implement `FileIO` if the environment supports throwing and catching the `IOError` exception.

Listing 3 gives one example of using this interface to implement `readFile`, throwing an exception if opening the file fails in `withFile`.

Listing 3. Implementing `readFile` via the `FileIO` interface

```
readFile : FileIO e => String -> App e String
readFile f = withFile f Read throw $ \h =>
               do content <- read [] h
                  pure (concat content)
where
  read : List String -> File -> App e (List String)
  read acc h = do eof <- fEOF h
                  if eof then pure (reverse acc)
                         else do str <- fGetStr h
                                 read (str :: acc) h
```

To implement `FileIO`, we need access to the primitive operations via `PrimIO`, and the ability to throw exceptions if any of the operations fail. With this, we can implement `withFile` as follows, for example:

[3]One could also imagine an interface using a linear resource for the file, which might be appropriate in some safety critical contexts, but for most programming tasks, exceptions should suffice.

```
Has [PrimIO, Exception IOError] e => FileIO e where
  withFile fname m onError proc
      = do Right h <- primIO $ openFile fname m
              | Left err => onError (FileErr (toFileEx err))
           res <- catch (proc h) onError
           pure res
  ...
```

Aside: The `| Left err =>` notation [Brady 2014] allows us to give alternatives to pattern matching bindings, as part of do-notation. A similar notation exists for let. This allows us to code to a default "happy path", providing alternatives for handling failure.

Given this implementation of FileIO, we can run readFile, provided that we wrap it in a top level handle function to deal with any errors thrown by readFile:

```
readMain : String -> App Init ()
readMain fname = handle (readFile fname)
       (\str => putStrLn $ "Success:\n" ++ show str)
       (\err : IOError => putStrLn $ "Error: " ++ show err)
```

4.5 Linear Resources with App

We have introduced App for writing interactive programs, using interfaces to constrain which operations are permitted, but have not yet seen the Path parameter in action. Its purpose is to constrain when programs can throw exceptions, to know where linear resource usage is allowed. The bind operator for App is defined as follows (not via Monad):

```
data SafeBind : Path -> (l' : Path) -> Type where
     SafeSame : SafeBind l l
     SafeToThrow : SafeBind NoThrow MayThrow

(>>=) : SafeBind l l' =>
        App {l} e a -> (a -> App {l=l'} e b) -> App {l=l'} e b
```

The intuition behind this type is that, when sequencing two App programs:

- if the first action might throw an exception, then the whole program might throw.
- if the first action cannot throw an exception, the second action can still throw, and the program as a whole can throw.
- if neither action can throw an exception, the program as a whole cannot throw.

The reason for the detail in the type is that it is useful to be able to sequence programs with a different Path, but in doing so, we must calculate the resulting Path accurately. Then, if we want to sequence subprograms with linear variables, we can use an alternative bind operator which guarantees to run the continuation exactly once:

```
bindL : App {l=NoThrow} e a ->
        (1 k : a -> App {l} e b) -> App {l} e b
```

This is similar to the variation of (>>=) we saw in Section 3.3, in that it is explicit that the continuation must run exactly once. To illustrate the need for bindL, we can try writing a variation of the data store which works in App, rather than IO. We use an interface for connecting and disconnecting:

```
interface StoreI e where
    connect : (1 prog : (1 d : Store LoggedOut) ->
```

```
                App {l} e ()) -> App {l} e ()
    disconnect : (1 d : Store LoggedOut) -> App {l} e ()
```

Neither `connect` nor `disconnect` throw, as shown by generalising over `l`. Once we have a connection, we can use the same interface as before, directly accessing the resource:

```
login : (1 s : Store LoggedOut) -> (password : String) ->
        Res Bool (\ok => Store (if ok then LoggedIn else LoggedOut))
logout : (1 s : Store LoggedIn) -> Store LoggedOut
readSecret : (1 s : Store LoggedIn) ->
             Res String (const (Store LoggedIn))
```

Listing 4 shows a complete program accessing the store, which reads a password, accesses the store if the password is correct and prints the secret data. It uses `let (>>=) = bindL` to redefine `do`-notation locally.

Listing 4. Access the data store, combining it with Console I/O

```
storeProg : Has [Console, StoreI] e => App e ()
storeProg = let (>>=) = bindL in
      do putStr "Password: "
         password <- getStr
         connect $ \s =>
           do let True @@ s = login s password
                | False @@ s => do putStrLn "Wrong password"
                                   disconnect s
              let str @@ s = readSecret s
              putStrLn $ "Secret: " ++ show str
              let s = logout s
              disconnect s
```

If we omit the `let (>>=) = bindL`, it will use the default `(>>=)` operator, which allows the continuation to be run multiple times, which would mean that `s` is not guaranteed to be accessed linearly, and `storeProg` would not type check. We can safely use `getStr` and `putStr` because they are guaranteed not to throw by the `Path` parameter in their types.

4.6 App1: Linear Interfaces

Adding the `bindL` function to allow locally rebinding the `(>>=)` operator allows us to combine existing linear resource programs with operations in `App`—at least, those that don't throw. It would nevertheless be nice to interoperate more directly with `App`. One advantage of defining interfaces is that we can provide multiple implementations for different contexts, but our implementation of the data store uses primitive functions to access the store.

To allow control over linear resources, we provide an alternative parameterised type `App1`:

```
data App1 : {default One u : Usage} ->
            (e : Environment) -> Type -> Type
```

There is no need for a `Path` argument, since linear programs can never throw. The `Usage` argument states whether the value returned is to be used once, or has unrestricted usage, with the default in `App1` being to use once:

```
data Usage = One | Any
```

The main difference from App is the (>>=) operator, which has a different multiplicity for the variable bound by the continuation depending on the usage of the first action:

```
Cont1Type : Usage -> Type -> Usage -> Environment ->
            Type -> Type
Cont1Type One a u e b = (1 x : a) -> App1 {u} e b
Cont1Type Any a u e b = (x : a) -> App1 {u} e b

(>>=) : {u : _} -> (1 act : App1 {u} e a) ->
        (1 k : Cont1Type u a u' e b) -> App1 {u=u'} e b
```

Cont1Type returns a continuation which uses the argument linearly, if the first App1 program has usage One, otherwise it returns a continuation where argument usage is unrestricted. Either way, because there may be linear resources in scope, the continuation is run exactly once and there can be no exceptions thrown.

Using App1, we can define all of the data store operations in a single interface, as shown in Listing 5. Each operation other than disconnect returns a *linear* resource.

Listing 5. The data store as an interface, where each App1 operation returns a linear resource

```
interface StoreI e where
  connect : App1 e (Store LoggedOut)
  login : (1 d : Store LoggedOut) -> (password : String) ->
          App1 e (Res Bool (\ok => Store (if ok then LoggedIn
                                                 else LoggedOut))
  logout : (1 d : Store LoggedIn) -> App1 e (Store LoggedOut)
  readSecret : (1 d : Store LoggedIn) ->
               App1 e (Res String (const (Store LoggedIn)))
  disconnect : (1 d : Store LoggedOut) -> App {l} e ()
```

We can explicitly move between App and App1:

```
app : (1 p : App {l=NoThrow} e a) -> App1 {u=Any} e a
app1 : (1 p : App1 {u=Any} e a) -> App {l} e a
```

We can run an App program using app, inside App1, provided that it is guaranteed not to throw. Similarly, we can run an App1 program using app1, inside App, provided that the value it returns has unrestricted usage. So, for example, we can write:

```
storeProg : Has [Console, StoreI] e => App e ()
storeProg = app1 $ do
     store <- connect
     app $ putStr "Password: "
     ?what_next
```

This uses app1 to state that the body of the program is linear, then app to state that the putStr operation is in App. We can see that connect returns a linear resource by inspecting the hole what_next, which also shows that we are running inside App1:

```
 0 e : List Type
 1 store : Store LoggedOut
-------------------------------------
what_next : App1 e ()
```

4.7 Implementation Details

Internally, `App` and `App1` work in the same way as `IO`, except that `App` supports exceptions, so needs to check whether an operation succeeded or failed. They are implemented as follows:

```
data App : (l : Path) => (e : Environment) -> Type -> Type where
  MkApp : (1 prog : (1 w : %World) ->
                    AppRes (execTy l e t)) -> App {l} e t

data App1 : (u : Usage) => (e : Environment) -> Type -> Type where
  MkApp1 : (1 prog : (1 w : %World) -> App1Res u t) -> App1 {u} e t
```

Both `AppRes` and `App1Res` correspond to `IORes` in the `IO` implementation, and the `(>>=)` operator for both is implemented similarly. While `App1Res` carries the result value directly, `AppRes` needs to calculate the result type from the exceptions allowed by the `Environment`:

```
data OneOf : Environment -> Path -> Type where
     First : e -> OneOf (e :: es) MayThrow
     Later : OneOf es MayThrow -> OneOf (e :: es) MayThrow

0 execTy : Path -> Environment -> Type -> Type
execTy p es ty = Either (OneOf es p) ty
```

Again, the `0` indicates that `execTy` is compile-time only. The `Path` index of `OneOf` statically guarantees that exception types are only available when an `App` can throw; this is useful when implementing `bindL`, to ensure that the initial action cannot fail. Implementing `App` and `App1` in this way minimises the overhead; indeed, since the types, path and usage indices are guaranteed to be erased, there is no overhead beyond `IO` other than error checking.

5 EXAMPLE: DEPENDENT SESSION TYPES

To illustrate how we can use `App` and quantitative types on a more substantial example, let us consider how to use them to implement session types. Session types [Honda 1993; Honda et al. 2008] 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 in functional programming[4]. 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.

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:

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

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

data Channel : Actions -> Type
```

Internally, `Channel` contains a message queue for bidirectional communication. Listing 6 shows an interface 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 6. Interface for initiating and executing concurrent sessions

```
interface Session e where
  send : (1 chan : Channel (Send ty next)) -> (val : ty) ->
         App1 e (Channel (next val))
  recv : (1 chan : Channel (Recv ty next)) ->
         App1 e (Res ty (\val => Channel (next val)))
  close : (1 chan : Channel Close) -> App1 {u=Any} e ()
  fork : (forall e' . PrimIO e' =>
              (1 chan : Server {a} p) -> App e' ()) ->
         ((1 chan : Client {a} p) -> App e a) -> App e a
```

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 7) in which a client sends a `Command` to either `Add` a pair of `Int`s or `Reverse` a `String`.

`Protocol` is a DSL for describing communication patterns. Embedding it in a dependently typed host language gives us dependent session types for free. 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) >>= \_ =>
```

Listing 7. A global session type describing a protocol where a client can request either adding two Ints 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
```

```
                    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 : (forall e' . PrimIO e' =>
            (1 chan : Server {a} p) -> App e' ()) ->
       ((1 chan : Client {a} p) -> App e a) -> App e a
```

The type of `fork` ensures that the client and the server are working to the same protocol, and, as with the primitive `fork`, that any `State` cannot be shared between threads since they run in different environments.

Listing 8. An implementation of a server for the Utils protocol

```
utilServer : Has [Console, Session] e =>
             (1 chan : Server Utils) -> App e ()
utilServer chan = app1 $
    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 8 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 effective 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 : Has [Console, Session] e =>
             (1 chan : Server Utils) -> App e ()
utilServer chan = ?utilServer_rhs

 0 e : List Type
 1 chan : Channel (Recv Command (\res => ... ))
------------------------------------
utilServer_rhs : App e ()
```

Therefore, the first action on `chan` must be to receive a `Command`. Furthermore, we are in `App`, and `recv` is in `App1` since the operations are linear, so we use `app1` to move into `App1`:

```
utilServer chan = app1 $
    do cmd @@ chan <- recv chan
       ?utilServer_rhs

   0 e : List Type
     cmd : Command
   1 chan : Channel (ServerK (case cmd of ...))
  ------------------------------------
  utilServer_rhs : App1 e ()
```

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 : Has [Console, Session] e =>
             (1 chan : Server Utils) -> App e ()
utilServer chan = app1 $
    do cmd @@ chan <- recv chan
       case cmd of
            Add => ?process_add
            Reverse => ?process_reverse
```

Again, we make essential use of dependent `case`, in that both branches have a different type which is computed from the value of the scrutinee `cmd`. 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`, we see we have to receive a pair of `Int`s, then send an `Int`:

```
   0 e : List Type
   1 chan : Channel (Recv (Int, Int) (\res =>
                    (Send Int (\res => Close))))
     cmd : Command
  ------------------------------------
  process_add : App1 e ()
```

Developing the server in this way shows programming as 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.

6 RELATED WORK

Substructural Types. Linear types [Wadler 1990] and other substructural type systems have been shown to have several applications, e.g. verifying unique access to external resources [Ennals et al. 2004] and as a basis for session types [Honda 1993]. 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 [de Vries et al. 2008; Morris 2016; Orchard et al. 2019; Tov and Pucella 2011]. Our work differs from a proposed extension to Haskell supporting linear types [Bernardy et al. 2017] in that, since Idris does not support exceptions as part of its run time system, we can more clearly express the relationship between linearity and exceptions in library code, e.g. with `App1`. While these languages do not have full dependent types, Granule [Orchard et al. 2019] allows many of the same properties to be expressed with a sophisticated notion of graded types which allows quantitative reasoning about resource usage. ATS [Shi and Xi 2013] 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[5] [Jung et al. 2017] which guarantees memory safety of imperative programs without garbage collection or any run time overhead, and is expressive enough to implement session types [Jespersen et al. 2015].

Historically, combining linear types and dependent types in a fully general way—with types as first class, 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 [Krishnaswami et al. 2015] by disallowing dependent linear functions or by limiting the form of dependency [Gaboardi et al. 2013], but these approaches limit expressivity. For example, we may still want to reason about linear variables which have been consumed. Recent work on Quantitative Type Theory [Atkey 2018; McBride 2016] which forms the core of Idris 2, 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 clear potential benefits in allowing reasoning about effects and resource usage, one of the main motivations for using QTT in the core of Idris 2 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" [Brady 2005], which give a predictable, if not fully general, method for determining which arguments can be erased. Idris 1 uses a whole program analysis [Tejiscak 2020], partly inspired by earlier work on Erasure Pure Type Systems [Mishra-Linger and Sheard 2008] 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 [Weirich et al. 2019]. Our experience of the 0 multiplicity of QTT so far is that it provide the cleanest solution to the erasure problem, although we can no longer infer which other arguments can be erased.

[5] https://rust-lang.org/

Interactive Editing and Program Synthesis. We have briefly discussed how QTT improves support for program synthesis by taking usage restrictions into account in the search. Program synthesis in Idris has not yet been explored deeply, and existing work on type-driven program synthesis [Polikarpova et al. 2016], resource-guided program synthesis [Knoth et al. 2019] and example-driven program synthesis [Osera and Zdancewic 2015] will provide important insight into improving program search. Nevertheless, even a brute force search of a hint database has (anecdotally) proved remarkably effective for small search problems.

There is also a lot of scope for using quantitative types in interactive editing support. To make linear dependent types practically useful and accessible to application developers, good interactive tooling is essential. Recent work on front end tooling [Robert 2018] and structural editing with typed holes [Omar et al. 2019] will influence Idris 2.

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 resource usage—and hence, reasoning about interactions with external libraries. Previous work on reasoning about effects and resources with dependent types has relied on indexed monads [Atkey 2009; McBride 2011] or embedded DSLs for describing effects [Brady 2014]. 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 to reasoning about effects and resource usage such as Hoare Type Theory [Nanevski et al. 2008] are sufficiently expressive, but difficult to apply in everyday programming.

The `App` library provides similar expressivity to runners of algebraic effects [Ahman and Bauer 2019], which provide a mathematical model of resource management, and, like our `(>>=)` operator, ensure that continuations are run at most once. While our approach using `App` is not as expressive as, say, algebraic effects [Lindley et al. 2017; Plotkin and Pretnar 2009], monad transformers [Liang et al. 1995] or separation logic, it is composable with these more expressive approaches in exactly the same way as `IO`. For example, `App` could be used at the bottom of a monad transformer stack, or as a way of instantiating a program built on algebraic effects or free monads.

Session Types. In Section 5 we gave an example of the application of QTT in implementing Dyadic Session Types [Honda 1993]. In previous work [Brady 2017] 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 [Thiemann and Vasconcelos 2019] with no additional cost. Previous work in exploring value-dependent sessions in a dependently typed language [de Muijnck-Hughes et al. 2019] is directly expressible using linearity in Idris 2. We have not yet explored further extensions to session types, however, such as multiparty session types [Honda et al. 2008], dealing with exceptions during protocol execution [Fowler et al. 2019] or dealing with errors in transmission in distributed systems.

7 CONCLUSION

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. However, 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, the potential of quantitative types in reducing the search space for type-driven program synthesis, and 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 `App` library is an example of this. It is valuable to an application programmer to be able to express whether a program throws an exception, and to say which interfaces a function needs. These are expressible only because the internals of the library manage the where exceptions can be thrown via dependent types, but these details are not visible to the programmer. So, while a library user may not experience much difficulty with a more limited form of dependent types, a library developer will!

7.1 Future work

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. 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 [Orchard et al. 2019], we may find it useful to use quantities other than `0` and `1`, and the theory behind QTT already supports this.

There is also scope for improvement in interactive editing tools. Since we have type-driven expression search in *holes* as well as for *complete definitions*, we can consider a constrained expression search, in domain specific contexts. For example, at each stage in a session type protocol, the next operation on the channel should be synthesisable from the type. Like program search, `auto`-implicit search would also benefit from a more rigorous theoretical treatment, perhaps following the Calculus of Coherent Implicits [Schrijvers et al. 2019].

Interactive editing with holes helps significantly when writing a program, but as yet offers little or no benefit during maintenance. This is significant, since most of a program's life is spent in maintenance. If we change a `Protocol`, for example, this will introduce type errors in the client and server. Refactoring tools to support this kind of update will be challenging to build, but type-driven editing should support refactoring at least to some extent.

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 [Kovács 2019], Idris 2 minimises substitution of unification solutions. Initial results are promising: although Idris 2 is not yet fully self hosting, it type checks its own core in 12 seconds, compared to the 50 seconds taken by Idris 1.

Finally, an important application of reasoning about linear resource usage is in implementing communication and security protocols correctly. The session type in Section 5 provides a preliminary example which demonstrates the possibilites, 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

Danel Ahman and Andrej Bauer. 2019. Runners of Algebraic Effects. Submitted.

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

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

Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS 2018*. https://doi.org/10.1145/3209108.3209189

Jean-Philippe Bernardy, Mathieu Boespflug, Simon Newton, Ryan R. an d Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

Edwin Brady. 2005. *Practical Implementation of a Dependently Typed Functional Programming Language*. Ph.D. Dissertation. University of Durham.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (9 2013), 552–593. Issue 05.

Edwin Brady. 2014. Resource-dependent Algebraic Effects. In *Trends in Functional Programming (TFP '14) (LNCS)*, Jurriaan Hage and Jay McCarthy (Eds.), Vol. 8843. Springer.

Edwin Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Computer Science* 18, 3 (2017).

Nils Anders Danielsson. 2010. Total parser combinators. In *International Conference on Functional Programming*, Vol. 45. 285. https://doi.org/10.1145/1932681.1863585

Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. 2019. Value-Dependent Session Design in a Dependently Typed Language. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019 (EPTCS)*, Francisco Martins and Dominic Orchard (Eds.), Vol. 291. 47–59. https://doi.org/10.4204/EPTCS.291.5

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

Robert Ennals, Richard Sharp, and Alan Mycroft. 2004. Linear Types for Packet Processing. In *ESOP*.

Simon Fowler, Sam Lindley, J Garrett Morris, and Sara Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. In *Principles of Programming Languages*, Vol. 3.

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR 19938 (International Conference on Concurrency Theory)*. Springer.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL 2008*.

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

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 253–268. https://doi.org/10.1145/3314221.3314602

András Kovács. 2019. Fast Elaboration for Dependent Type Theories.

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

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM Press, New York, New York, USA, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Principles of Programming Languages*. arXiv:arXiv:1611.09259v1

Conor McBride. 2011. Kleisli arrows of outrageous fortune.

Conor McBride. 2014. How to Keep Your Neighbours in Order. In *International Conference on Functional Programming*.

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

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *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 (Lecture Notes in Computer Science)*, Roberto M. Amadio (Ed.), Vol. 4962. Springer, 350–364. https://doi.org/10.1007/978-3-540-78499-9_25

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

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

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

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 15)*. Association for Computing Machinery, New York, NY, USA, 619630. https://doi.org/10.1145/2737924.2738007

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

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*. 80—-94.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs*. Ph.D. Dissertation. University of California San Diego.

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. *J. Funct. Program.* 29 (2019), e3. https://doi.org/10.1017/S0956796818000242

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

Matus Tejiscak. 2020. *Erasure in Dependently Typed Programming*. Ph.D. Dissertation. University of St Andrews.

Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL, Article Article 67 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371135

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

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

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

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