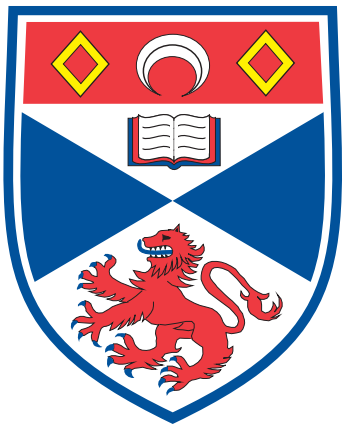
Verified Networking using Dependent Types



University
of
St Andrews

CS4099: Major Software Project

Simon Fowler

2014-04-04

Supervisor: Dr Edwin Brady

Abstract

Strongly, statically typed functional programming languages have found a strong grounding in academia and industry for a variety of reasons: they are concise, their type systems provide additional static correctness guarantees, and the structured management of side effects aids easier reasoning about the operation of programs, to name but a few.

Dependently-typed languages take these concepts a step further: by allowing types to be predicated on values, it is possible to impose arbitrarily specific type constraints on functions, resulting in increased confidence about their runtime behaviour.

This work demonstrates how dependent types may be used to increase confidence in network applications. We show how dependent types may be used to enforce resource usage protocols inherent in C socket programming, providing safety guarantees, and examine how a dependently-typed embedded domain-specific language may be used to enforce the conformance of packets to a given structure. These concepts are explored using two larger case studies: packets within the Domain Name System (DNS) and a networked game.

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 19,993 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

Chapter 1

Introduction

Functional programming languages with strong, static type systems are becoming increasingly popular due to the host of benefits they offer. Purely-functional languages allow side-effects to be managed in a pure way, aiding reasoning about programs and facilitating concurrent and parallel programming. The strong, static type systems offered by these languages enable more errors to be caught at compile time, increasing static guarantees about correct runtime behaviour. Programs can be expressed in a concise and high-level way, meaning that codebases are more maintainable and understandable. The list continues.

Concentrating on correctness guarantees, we may use *dependent types* to enforce even stronger specifications. Languages with dependent types allow types to be *predicated on values*, meaning that arbitrarily specific type signatures may be introduced. By using more specific types to encode invariants about programs, we add program verification at the language level, resulting in correct-by-construction programs. Traditionally, however, dependently-typed languages have been largely only been used as theorem-proving tools, which results in a host of missed opportunities: in particular, errors that may be caught statically as a result of more specific types instead cause a runtime error in a deployed system.

1.1 Problem Outline

Since dependently-typed languages have not been widely used as general-purpose programming languages, little investigation has as yet been done into how dependent types may be used for increasing the confidence in network applications.

In this work, we aim to firstly provide the basic infrastructure on which networked applications may be built, using recent research developments such as dependent algebraic effects [4] to show how resource usage protocols may be enforced.

We also reimplement and extend an embedded domain-specific language (EDSL) for specifying the structure of packets, and statically verifying the adherence of packet implementations to these specifications.

1.1.1 Contributions

The contributions of this project are as follows:

- A high-level binding to the C sockets library.
- Libraries for TCP and UDP, making use of dependent algebraic effects to enforce resource usage protocols and failure handling.

- An EDSL `PacketLang` specifying packet structure, and data between high- and low-level representations.
- An effectual binding to the Idris message-passing concurrency system
- Two larger case studies showing how the libraries may be used in larger applications: one using the `PacketLang` EDSL to implement a library to encode and decode DNS packets; and another implementing a networked game.

Chapter 2

Objectives

The overarching aim of this project is to investigate how dependent types may be used within the domain of network programming. Building upon previous work, the aim is to investigate, using current research into general-purpose dependently-typed programming, the ways in which dependent types may be used within the domain of network programming.

Deliverable 1 (Description, Objectives, Ethics and Resources) set out the objectives for the project as follows:

2.0.2 Primary

- A framework for the creation of provably correct application-layer protocols, built on top of TCP.
- Verified network bindings to underlying TCP libraries, enforcing correct usage.
- Sample applications making use of the network bindings.
- A verified implementation of the DNS protocol, including support for label compression

2.0.3 Secondary

- Investigation into how better to handle raw binary data efficiently within a dependently-typed language.
- Investigation into an extension of the current foreign function interface to better handle raw binary data.

2.0.4 Tertiary

- A framework for the creation of provably-correct transport layer protocols, built on top of IP.
- Support for threading, allowing the creation of more complex network applications.
- Investigation into how to measure and mitigate the overheads imposed by the verified bindings.

As the project was reasonably loosely-defined to begin with, the objectives consisted of possible avenues of exploration. After beginning work on the project, it soon became apparent that completing all of these objectives was far too ambitious given the time available, and that concentrating on the primary goals were more appropriate for the scope of the project. In particular, these were refined to:

- To implement an low-level Idris interface to the C sockets API, `Network.Socket`.
- To implement verified TCP and UDP bindings using the `Network.Socket` library.
- To reimplement the `PacketLang` DSL [7], improving it if necessary.
- To implement a larger case study, the Domain Name System (DNS), to assess the expressiveness of `PacketLang`.

Chapter 3

Context Survey

3.1 The Idris Programming Language

Idris [5] is a purely functional programming language with full-spectrum dependent types. In contrast to other systems such as Agda [29] and Coq [2], Idris places a large emphasis on being a *general-purpose* language, in particular being suitable for verified systems programming using a dependently-typed foreign-function interface (FFI). Additionally, efficiency is also a primary concern of Idris, in particular making use of aggressive type erasure [6] to mitigate the overheads of storing redundant type information.

To demonstrate the concept of dependent types, consider the example of a function which reverses a list, where `a` is a polymorphic type variable constrained by the `Ord` type class which provides a comparator function.

```
reverse : Ord a => [a] -> [a]
```

This type signature enforces several invariants on the resulting implementation. In particular, we know that the function takes one argument, a list of type `a`, and returns a list of the same type. If, for example, the argument given to a function was of type `[Int]`, attempting to return a value of type `[String]` would result in a compile-time error.

At the same time, an implementation which always returns the empty list `[]` would conform to this type specification and therefore correctly type-check, even though it is semantically incorrect.

Using dependent types, we may refine the specification to demand that the lists are of the same length. `Vect n a` is a list of length `n` of type `a`, where `n` is a type-level Peano natural number.

```
Ord a => Vect n a -> Vect n a
```

This refinement provides us with additional confidence about the correctness of the implementation of the function. Of course, it is still entirely possible to write implementations which satisfy the specification but are semantically incorrect. We may further increase the specificity of the type signature, requiring some proof `Reversed` that the output list is the reverse of the input list.

```
Ord a => (in : Vect n a) -> (Reversed in out ** (out : Vect n a))
```

The `(a ** b)` notation denotes a Σ-type or dependent pair, meaning that the second argument of the pair depends on the first.

Idris syntax is heavily inspired by Haskell, but with several subtle differences: in keeping with conventions in the literature, `:` is used to denote membership of a type, whereas in Haskell it is used to denote the `cons` operation of a list.

3.1.1 Type-level Computation

A powerful feature of dependently-typed languages is the ability to compute types as *first-class* terms. Through the use of this, we may construct a variadic adder which first takes the number of arguments to add `n`, and then requires `n` arguments to be specified. In order to do this, we firstly inductively define the Peano natural numbers (included in the Idris standard library):

```
data Nat = Z
         | S Nat
```

We next define a function `adderTy` which takes an input `(n : Nat)` and inductively defines appropriate function type:

```
adderTy : (n : Nat) -> Type
adderTy Z = Nat -> Nat
adderTy (S k) = Nat -> (adderTy k)
```

Finally, we implement the function `variadic` which is of type `(n : Nat) -> (adderTy n)`. The notation `(n : Nat)` denotes that the argument with type `Nat` is named `n` and may be used in further computations. More formally, this corresponds to a Π type in dependent type theory.

```
variadic : (n : Nat) -> (adderTy n)
variadic Z x = x
variadic (S k) x = \y => x + (variadic k y)
```

The `variadic` function captures the number of arguments and the first numerical argument on the left-hand side of the equation. In the base case, there are no additional numbers to add, and therefore no free variables to be captured on the right-hand side, and thus the captured variable is returned. In the inductive case, we capture the remaining free variables via an anonymous function on the right-hand side (similar to Haskell, `\` denotes a λ-abstraction in Idris), and use them in the recursive step.

Type-level computation is also extremely useful when dealing with *universes*: a data type which defines a set of tags representing types. These may then be translated into Idris types through a translation function, a technique demonstrated by Oury and Swierstra [30].

```
data Univ = INT | BOOL | STRING | CHAR

interpUniv : Univ -> Type
interpUniv INT = Int
interpUniv BOOL = Bool
interpUniv STRING = String
interpUniv CHAR = Char
```

3.1.2 (Embedded) Domain-Specific Languages

A large emphasis in the Idris programming language is placed on usability, with features such as overloading of constructs such as `do`-notation and idiom brackets [26], and syntax macros. Through the use of these features, it becomes much easier to implement *Embedded Domain-Specific Languages* (EDSLs), which provide an abstraction over more complex underlying types to allow developers to write higher-level domain-specific code.

Domain-specific languages are languages which are designed to aid programming for a particular domain, such as SQL for database programming. Whilst losing expressivity—most DSLs are not

Turing-complete, for example—they allow developers to more precisely and concisely handle domain-specific tasks. EDSLs are DSLs which are embedded in a *host language*, and may therefore make use of the type systems and constructs provided by the language.

3.1.3 Effect Handling

3.1.3.1 Monads and Monad Transformers

Purely functional languages are extremely useful for concisely expressing otherwise computations, and allowing powerful reasoning to take place about the correctness and operation of programs. This alone, however, does not make a language widely usable: for a language to be useful, it must be able to interact with its environment, causing *side effects*.

Haskell, and indeed Idris, solve this through the use of *monads* [32]. Monads are a structure with roots in category theory, and allow for impure, side-effecting computations to be modelled in a purely functional language.

Although more complete explanations may be found elsewhere [36], briefly, a monad is a typeclass providing at least two functions:

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

The `return` function takes a pure value and lifts it into a monadic context. The second operation $(>>=)$ (pronounced *bind*) takes a value `a` in a monadic context `m`, and a function taking `a` and producing a value `m b`.

To handle side effects, programs in these languages are given an entry point, `main : IO ()`, where `IO` is a monad supporting impure computations. In both languages, `IO` is handled as a special case, with language-based primitives being used to handle monadic binding operations.

To combine multiple monads (for example, if we wished to perform I/O operations and also retain some state), we may define a *monad transformer* [21], which allows the composition of two different monads. A monad transformer consists of an outer monad and an inner monad; to perform an operation in the context of the inner monad, the `lift` function is used. These suffice for very small numbers of monads, but soon become unwieldy when larger numbers are used. This leads library developers to either permit operations which may perform arbitrary IO actions (which may break abstractions and safety), or to combine many effects code into a coarse-grained monad.

3.1.3.2 No More Heavy Lifting: Resource-Dependent Algebraic Effects

More recent work focuses on handling effects *algebraically* [31]. Making use of the handler abstraction advocated by Kammar et al. [18], recent work has made use of dependent types to implement a framework, `Effects` [4], to facilitate programming with algebraic effects. This has several advantages: effects are *composable*, meaning that we may specify multiple effects and use them without needing to explicitly include `lift` calls; and they allow a developer to specify an associated resource. Since operations may be predicated on a certain resource type, and this resource may change after an operation, `Effects` makes it much simpler to enforce resource usage protocols.

Further work on `Effects` has brought substantial changes to the framework since its initial publication. For brevity, we outline the latest version here.

We begin with one of the canonical examples of enforcing resource usage protocols in dependently-typed programming: accessing a file[1]. We want to allow files to be opened for either reading or

[1] We make use of the implementation included in the main Idris distribution for this example.

writing, and ensure that read and write operations only happen when in the correct mode. We also want to ensure that all resources are released once the file handle is no longer needed.

To do this, we firstly describe the File IO operation abstractly by defining a generalised algebraic data type of kind `Effect`.

```
Effect : Type
Effect = (x : Type) -> Type -> (x -> Type) -> Type

data FileIO : Effect where
  Open  : String -> (m : Mode) ->
          {() ==> {res} if res then OpenFile m else ()} FileIO Bool
  Close : {OpenFile m ==> ()}               FileIO ()

  ReadLine  :           {OpenFile Read}  FileIO String
  WriteLine : String -> {OpenFile Write} FileIO ()
  EOF       :           {OpenFile Read}  FileIO Bool
```

The main difference between the latest `Effects` library and the published version is the fact that the output resource may be calculated from the result of the operation, which greatly helps with failure handling. The `Effect` kind specifies three parameters: a result type, an input resource type, and a function from the result type to an output resource type. To aid developers in specifying the types of effectual operations, `Effects` provides syntactic sugar. The `Open` operation provides an example:

```
 Open  : String -> (m : Mode) ->
         {() ==> {res} if res then OpenFile m else ()} FileIO Bool
```

The operation takes two parameters: a filename and a `Mode m` (an ADT specifying file modes). The effect type specifies that the operation requires an uninitialised resource `()`, and returns a `Bool`. If this is true, then the output resource is `OpenFile m`, whereas if it failed then the output resource remains uninitialised.

In order to use an effect, we must do two further things: specify *handlers*, which define how each effectual operation may be interpreted in a given execution context, and promote it to a *concrete* effect.

Handlers are defined by making an effect and an execution context an instance of the multi-parameter `Handler` type class. Here, `m` is an execution context, which is often a monad but this does not always have to be the case: for example, to handle an effect in a pure context, `id` is sufficient.

```
class Handler (e : Effect) (m : Type -> Type) where
     handle : res -> (eff : e t res resk) ->
              ((x : t) -> resk x -> m a) -> m a
```

Here, `res` is the input resource, and `eff` is the effect. The final argument is a continuation function which takes the result and value of the output resource. We must then specify `handle` functions for each operation, pattern matching on the resource to make use of the file handle if it exists, and pattern matching on the abstract effect operation to make use of the parameters.

```
instance Handler FileIO IO where
    handle () (Open fname m) k = do h <- openFile fname m
                                    valid <- validFile h
                                    if valid then k True (FH h)
                                             else k False ()
    handle (FH h) Close      k = do closeFile h
```

```
                                  k () ()
    handle (FH h) ReadLine        k = do str <- fread h
                                         k str (FH h)
    handle (FH h) (WriteLine str) k = do fwrite h str
                                         k () (FH h)
    handle (FH h) EOF             k = do e <- feof h
                                         k e (FH h)
```

To use these in operations, we must promote the abstract effect into a concrete effect, which is achieved using the `MkEff` function. The `Type` parameter denotes the resource type.

```
FILE_IO : Type -> EFFECT
FILE_IO t = MkEff t FileIO
```

To make working with effects easier, we then create wrapper functions which may be used in effectual programs. The wrapper function for the `Open` operation is shown below.

```
open : Handler FileIO e =>
       String -> (m : Mode) ->
       { [FILE_IO ()] ==>
         [FILE_IO (if result then OpenFile m else ())] }
       Eff e Bool
open f m = Open f m
```

Finally, we may write a program making use of this effect, and execute it in the underlying execution context using the `run` function.

```
fileTest : String -> {[FILE_IO ()]} Eff IO (Maybe String)
fileTest fn = with Effects do
  True <- open fn Read
    | False => return Nothing
  line <- readLine
  close
  return (Just line)

main : IO ()
main = do
  m_line <- run (fileTest "test_file.txt")
  case m_line of
    Just line => putStrLn $ "First line: " ++ line
    Nothing => putStrLn "Error opening file for reading"
```

Note in particular the guard syntax in `fileTest`: in this notation, we specify the expected result of `open` as True, assuming that the file was successfully opened. We handle possible failure cases inline using guard notation. The code following the guard notation is the code which is executed in the success case.

3.2 Related Work

3.2.1 Haskell with Language Extensions

While the core type system of Haskell is different from that of a language with full-spectrum dependent types (in particular, lacking full Π-types), recent work has steered Haskell's type system

towards one supporting ever more dependently-typed features. In particular, the Glasgow Haskell Compiler (GHC) supports Generalised Algebraic Data Types (GADTs) [17], which are a powerful tool for allowing matching to refine type variables; the `DataKinds` extension [40] which allows the promotion of data types to the kind level; and type families, which allow limited type-level computation. The Strathclyde Haskell Enhancement [25] additionally provides a preprocessor to simulate dependent types within Haskell.

At the same time, Haskell provides no mechanisms for first-class theorem proving, much less the more powerful tactic-based approaches used in Coq and Idris. As outlined by Lindley and McBride [22], Haskell requires many language extensions and encoding of otherwise trivial lemmas to accomplish tasks which would be relatively simple in other languages. Conversely, they show that GHC's constraint solver may be used in some circumstances to avoid explicit calls to some lemmas once they are defined.

3.2.2 Data Description Languages

The challenge of representing binary data is not a new one, and several approaches exist for specifying binary data as embedded domain specific languages. In this section, we discuss some of the existing solutions to this problem, and contrast them with the `PacketLang` language used within this project.

3.2.2.1 PacketTypes

PacketTypes [27] is a standalone domain-specific language which allows the declarative specification of packet structures. As well as specifying and naming primitive fields, PacketTypes allows constraints to be placed on the data in a separate `where` clause. Other functionality included within PacketTypes involves the notions of *overlaying*: that is, embedding one packet type within another, and *refinement*, which adds additional constraints to an existing packet.

The approach of using *types* to specify packet descriptions is a common theme between PacketTypes and the `PacketLang` DSL. Our language allows the possibility of constraints on data through the `Proposition` constructs (which are internally implemented using a Σ type binding approach), but this is done as part of the packet specification as opposed to being in a separate clause. Overlaying is easily supported in `PacketLang` as each packet specification may be bound as a first-class construct, and refinement may be achieved simply by specifying additional constraints.

The main difference between the two systems is that PacketTypes is a *standalone* DSL, meaning that it has its own compiler and type system. `PacketLang`, on the other hand, is an *embedded* domain specific language, meaning that it makes use of the type system of the host language. Whereas PacketTypes works by generating parsing and marshalling code, `PacketLang` is completely defined within Idris itself. Another limitation of PacketTypes is the fact that the interpretation of later parts of packets may not depend on the values of data specified earlier in the packet. This would make it unable to represent DNS packets, for example.

3.2.2.2 The Data Description Calculus

The Data Description Calculus (DDC) [11] is a formalism for describing data definition languages using a small core dependent type theory, and provides a formal semantics for other data description languages. We do not attempt a formal translation from `PacketLang` into the DDC, but many of the constructs and semantics of `PacketLang` correspond to the DDC.

3.2.2.3 Protege

Protege [39] is a domain-specific language embedded in Haskell, primarily geared towards specifying protocol stacks in embedded systems such as sensor networks. Using Protege, developers may specify not only packet structures, but also protocols and protocol stacks: that is, the order in which packets may be sent and received, and how protocols interact. Although the authors' treatment of protocols has little in the way of formal grounding in the context of session types, their finite state machine approach provides a sufficient way of specifying packet ordering for the purposes of the implementation.

As an embedded domain-specific language, Protege differs from PacketTypes in that it does not require an external compiler, instead making use of the constructs present in Haskell itself. Similar to PacketTypes and differently to `PacketLang`, however, Protege operates by generating C parser code instead of working with decoded data in Haskell itself. This is due to the application domain of embedded systems, which operates almost exclusively with C.

Protege makes use of the constructs defined within the DDC.

3.2.2.4 DataScript

DataScript [1] is a data description language similar to PacketTypes. By using Java as a target language, class files are generated which contain accessor methods for each field of the data representation, and provide a constructor to marshal Java code into this representation. In this sense, it may be seen as similar to middleware such as CORBA [35], with the important difference that DataScript is designed to specify the exact physical layout of the marshalled data.

Once again, DataScript is a standalone DSL, requiring an external compiler. It also does not provide parser-like primitives such as choice, unlike `PacketLang`.

3.2.2.5 PADS

PADS [10] is another data description language which again takes the standalone DSL approach: descriptions may be compiled to generate either parsing and parsing code, or code to convert the described ad-hoc data definitions into more standard formats such as XML.

3.2.2.6 A Library for Processing Ad hoc Data in Haskell

Wang and Gaspes [38] describe a data description language embedded in Haskell. Unlike Protege, DataScript and PADS/ML and more similarly to `PacketLang`, this library is an *embedded* DSL and as such does not involve an additional compiler or code generation.

The library is an embedding of the DDC in Haskell, and makes heavy use of the ideas behind monadic parser generators such as Parsec [19]. This technique, unlike that of `PacketLang`, is unidirectional: that is, while it allows data to be parsed in, it provides no facility for writing data back out according to a specification. We also gear our implementation more towards packet data, providing functions specific to the marshalling and unmarshalling of packet data from network sockets.

3.2.3 Previous Work

This work is largely inspired by the paper "Idris—Systems Programming Meets Full Dependent Types" [7], which motivates Idris as a high-level language capable of being used for low-level applications. In particular, the main claim made is that dependent types may be used to provide additional static guarantees about low-level systems code, through the use of domain-specific languages and a dependently-typed foreign function interface.

In this work, we reimplement and extend the `PacketLang` EDSL for specifying binary packet data within Idris, and marshalling and unmarshalling data using the packet specifications.

Development of the Idris language is rapid. In the three years since the publication of this paper, the language has been completely rewritten, making the existing implementation of `PacketLang` obsolete beyond simple repair. Additionally, new language features such as type classes and the `Effects` framework provide interesting avenues of exploration for improving the existing implementation.

Recent previous work has concentrated on using `Effects` to implement resource usage protocols for web applications including a database library and the Common Gateway Interface (CGI) [12]. We build upon the lessons learnt from this work, in particular with respect to failure handling, in the task of enforcing resource usage protocols for TCP and UDP.

Chapter 4

Requirements Specification

4.1 C Socket Bindings

Socket Creation
Users should be able to obtain a C socket descriptor, given socket parameters.

Socket Abstraction
The socket descriptor should be wrapped in another data type to preserve type-safety. That is, socket descriptors should not be exposed as an integer, even though that is their underlying representation.

Socket Binding
Users should be able to bind a created socket to a socket address and port.

Listening on a socket
Users should be able to listen on a bound stream socket.

Accepting Clients
Users should be able to accept incoming connections from a bound stream socket.

Closing Sockets
Users should be able to close sockets, in turn releasing the held resources.

Error Reporting
Errors should be reported as part of the result of the operation, and not require an additional top-level function call. That is, errors should be exposed as a sum type with the result of a successful operation.

Sending Data
Users should be able to send data to a remote host using a connected socket in the case of a stream socket, or to a given address and port in the case of a datagram socket. There should be two modes for sending data: one for sending a string of data, and one for sending a populated raw buffer.

Errors with sending or receiving data should be reported.

Receiving Data
Users should be able to receive data from a connected stream socket or from a bound datagram. There should be two modes for receiving data: one for receiving a string of data, and one for populating a buffer.

In the case of receiving from a datagram socket, the socket address of the remote host should also be returned.

4.2 TCP Bindings

Connection
Clients should be able to connect to a remote host, given a socket address and port.

Sending Data
Connected clients should be able to send data to remote host. If a user writes code which attempts to send data to either an unconnected socket, or has not checked whether a previous send or receive operation succeeded, this should be caught at compile time. If an error occurs when attempting to send data, further send and receive operations using the socket should not be permitted.

Receiving Data
Connected clients should be able to receive data from the remote host. As above, this should only be possible in a valid state, and code which does not guarantee this should not compile. If the read operation fails, further send and receive operations should not be permitted.

Closing the socket
Users should be able to close a socket and release all opened resources. By specifying a type signature demanding an uninitialised resource at the start and end of the function, and assuming a total function, all code paths should result in the associated resources being released. If this is not the case, then the user should be notified at compile time.

Socket Abstraction
Users should never have to explicitly specify a socket as a parameter to any operations within the effect: this should be handled by the effect itself.

Binding
TCP server sockets should be able to bind to an address and port.

Listening
Successfully-bound TCP server sockets should be able to listen for new connections. If a socket is not successfully bound and the user attempts to listen for connections, then a compile-time error should result.

Accepting Clients
Listening TCP server sockets should be able to accept new clients, either in the same thread or a new thread. The state of the client socket should not affect the state of the server socket.

Client Handler Function
When accepting a client, users should be able to specify a handler function to handle the incoming client. The type of this function should specify, assuming the function is total, that the socket should be closed once the function terminates.

Sending and Receiving Data
It should be possible to send data to, and receive data from, accepted client sockets, assuming that they are in a valid state. If a send or receive operation fails, then the socket should be invalidated.

4.3 UDP Bindings

Sending Data
Users should be able to asynchronously send data to another socket, given a socket address and port.

Socket Abstraction
Users should not have to manually obtain or specify socket descriptors: this should instead be handled by the relevant operations.

Binding
Users should be able to bind a UDP server to a given address and port.

Sending and Receiving Data
Users should be able to send and receive data on a *bound* server socket, and send data from a UDP client. Sending data to a remote host will require the socket address and port of the remote host, and receiving data from the bound socket will also return the socket address and remote port of the remote host.

It should not be possible to send or receive data on an unbound server socket, or a socket in an invalid state.

4.3.1 `PacketLang` DSL

DSL Data Types
Users should be able to specify bounded binary data, null-terminated strings, length-indexed strings, Boolean values, and propositions on data.

DSL Control Constructs
The DSL should include constructs for choice (both by testing on a Boolean predicate, and by testing two specifications), lists of data, length-indexed lists of data, and sequencing.

Data Dependencies
Packet specifications should be able to depend on previously-specified data. For example, the length of an integer may be specified by a length field.

Data Marshalling
If, and only if, a packet specifies a packet specification, it should be possible to marshal the data into a packet for transmission either by TCP or UDP. Attempting to construct a packet using an implementation which does not correspond to a given packet specification should result in a compile-time type error.

Data Unmarshalling
If, and only if, an incoming packet satisfies a packet specification, it should be possible to unmarshal the incoming data into its constituent components.

4.3.2 DNS Library

Requests
It should be possible for users to make DNS requests using a specified type and class.

Responses
It should be possible for users to receive responses from DNS servers in user-friendly, high-level data types.

Payload Implementations
Common payload types such as A and CNAME records should be supported.

Data Type Correctness
Payloads in higher-level Idris data types should only contain payloads which correspond to a given DNS type and class. It should not be possible to create an instance of a DNS packet with a payload not corresponding to the stated type and class combination. Length fields should correspond to the length of each section.

Packet Correctness It should not be possible to create a packet which does not conform to the DNS packet specification. This means that it should not be possible to create a packet with a payload that does not match the given type and class combination; all types and class values should be valid according to the specification; length fields should correspond to payload lengths; count fields should correspond to the number of records in each section; and all integers should fit within their relevant bounds.

Chapter 5

Development Methodology

5.1 Rapid Application Development

As this is a primarily a research project, the priority was largely to create working software quickly, in order to demonstrate the feasibility of research concepts. For this reason, rapid application development (RAD) [24] was chosen as the primary software development methodology.

Rapid application development makes use of an underlying iterative development methodology, as outlined in Figure 5.1.

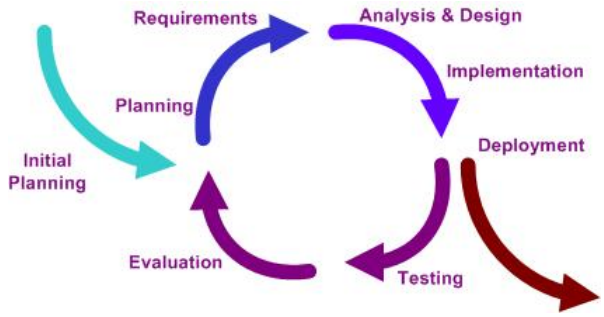


Figure 5.1: Iterative Development Model [33]

The short cycles enable software to be developed incrementally, with the evaluation of previous iterations being used in subsequent iterations. This works particularly well when developing research code, where certain concepts may prove to work successfully, or conversely certain methods may prove to be unsuccessful and require revisiting. The aim at the end of each cycle, however, is to have a working software system, and improve upon this in subsequent iterations.

Rapid application development incorporates iterative development with the added notion of developing prototypes. By building prototypes of software components (for example, the `PacketLang` DSL), we may use this as a base to test the ideas, revisiting and revising them as necessary if issues arise later in the process.

Figure 5.2 shows the main workflow within the rapid application development process. RAD encompasses four phases: planning, in which requirements are planned; user design, in which the system is used and additional requirements are elicited; construction, in which development takes place; and finally cutover, in which the new system is implemented.

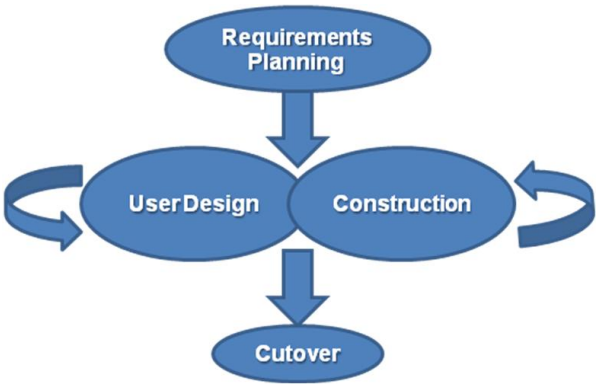


Figure 5.2: Rapid Application Development [34]

5.2 Type-Driven Development

The use of dependent types during development naturally fosters particular styles of development: whereas with a dynamically-typed language, it would be foolish to write a nontrivial piece software in anything but a test-driven style with extensive unit testing coverage, languages with more expressive type systems provide additional safety guarantees within the type system, reducing the need for unit testing. Of course, unit testing only detects the *presence* of defects, and doesn't guarantee their absence.

As well as this, types form an integral part of the development process. The nature of dependently-typed programming involves encoding specifications about the implementation of programs within type signatures. This runs contrary to certain trends within functional programming: users of ML-based languages such as OCaml [20], for example, rely on *type inference*: that is, inferring a type from an implementation, and raising errors if types have been inconsistently used within the implementation.

Type inference is, however, undecidable for languages such as Idris which incorporate full-spectrum dependent types. Whilst at times (for example, on inner functions declared in `where` clauses) this is undesirable, it can be viewed as a feature as opposed to a limitation: with specific enough types it becomes possible to infer *programs*, which has been recently incorporated into interactive editing tools [3].

Another useful tool made possible due to the Idris type system is the possibility to specify placeholders, or *metavariables*. By then entering proof mode, we may examine the context of the metavariable—that is, the names and types of other variables that are in scope—and use this information to guide the implementation. To take a concrete example, let us examine an incomplete implementation of the `encodeRR` function, which is used when encoding a DNS resource record into a format which may be written to a packet.

```
encodeRR : DNSRecord -> Either DNSEncodeError (mkTy dnsRR)
encodeRR (MkDNSRecord name ty cls ttl rel pl) = with Monad do
  dom <- encodeDomain name
  b_ttl <- isBounded 32 (intToNat ttl)
  encoded_pl <- encodePayload rel pl
  let pl_len = (bitLength (dnsPayloadLang ty cls) encoded_pl) `div` 8
  b_len <- isBounded 16 (intToNat pl_len)
  Right (dom ## ty ## cls ## b_ttl ## b_len ## encoded_pl ## ?mv)
```

In this code snippet, we are unsure of the type of the final argument needed to construct the packet implementation, and have thus inserted a metavariable `?mv`. We may then enter proof mode to retrieve the current context and goal type:

```
----------              Assumptions:              ----------
 pl_ty : DNSPayloadType
 name : List String
 ty : DNSType
 cls : DNSClass
 ttl : Int
 rel : DNSPayloadRel ty cls pl_ty
 pl : DNSPayload pl_ty
 class : Monad (Either DNSEncodeError)
 dom : mkTy dnsDomain
 class1 : Monad (Either DNSEncodeError)
 b_ttl : Bounded (fromInteger 32)
 class2 : Monad (Either DNSEncodeError)
 encoded_pl : mkTy (dnsPayloadLang ty cls)
 pl_len : Length
 class3 : Monad (Either DNSEncodeError)
 b_len : Bounded (fromInteger 16)
----------                  Goal:                  ----------
{hole16} : (\x5 =>
              val b_len =
              div (bitLength (dnsPayloadLang ty cls) x5) 8) encoded_pl
```

This tells us that the final argument is an equality proof between the value of the `b_len` argument, and the length of the encoded DNS payload.

5.3 Tools Used

From the outset, this project has been made publicly available on GitHub[1]. In any larger project, version control is of course essential, as it allows changes to be made from different working locations, provides an audit trail in case a regression should arise, and additionally provides an important avenue of code dissemination should other developers wish to use or contribute to the project.

[1] http://www.github.com/SimonJF/IdrisNet and later http://www.github.com/SimonJF/IdrisNet2.

Chapter 6

Ethics

This project made no use of human subjects or external data. As such, there were no ethical considerations for this project and accelerated ethical approval was sought.

Chapter 7

Design

7.1 System Overview

The system is split up into several components:

Socket Library
The socket library `IdrisNet.Socket` is an Idris binding to lower-level C socket functionality. It provides data types and bindings to functions in the C socket library, with the intention of allowing developers the flexibility to write additional applications on top of the standard interface.

C Helper Library
The socket library `idrisnet.c` makes use of the Idris foreign function interface to communicate with native code. In some instances, it is possible to call these functions directly from the Idris code, if there exist direct translations from the Idris types to the types in the C function. This is not possible in all cases however, and this helper library exists to facilitate this interaction by performing any additional setup and translations between types that is required.

TCP Server Effect
The TCP server effect `IdrisNet.TCP.TCPServer` is a module allowing developers to write TCP servers that are guaranteed to conform to correct resource usage protocols.

TCP Client Effect
The TCP client effect `IdrisNet.TCP.TCPClient` is a module allowing developers to connect to, and communicate with, a TCP server.

UDP Server Effect
The UDP server effect `IdrisNet.UDP.UDPServer` is a module allowing developers to bind to a UDP socket, and send and receive data using UDP.

UDP Client Effect
The UDP client effect `IdrisNet.UDP.UDPClient` exists as a stateless UDP client, allowing users to send UDP data to other UDP sockets.

Packet DSL
The Packet DSL module `IdrisNet.PacketLang` provides a domain-specific language for specifying the format of packets. Making use of dependent types, we may use this DSL to ensure that packets conform to the packet descriptions.

Packet Library
The packet library `IdrisNet.Packet` uses the packet descriptions to marshal the contents into a buffer to be sent. Additionally, we may use packet descriptions to unmarshal packets into their constituent components for use in a user application.

Process Effect
The Process library `Effect.Process` provides an effectual interface to the existing message-passing concurrency functionality, along with new abstractions for allowing spawned threads to operate with effects.

Figure 7.1 gives a high-level overview of the system, showing the interactions between components. In particular, the socket library interacts with the underlying C helper library, which itself interacts with the C sockets API. User programs may interact with the Packet DSL in order to construct packets and interpret packet definitions, but do not interact directly with the packet library. This functionality is instead exposed by the protocol bindings.

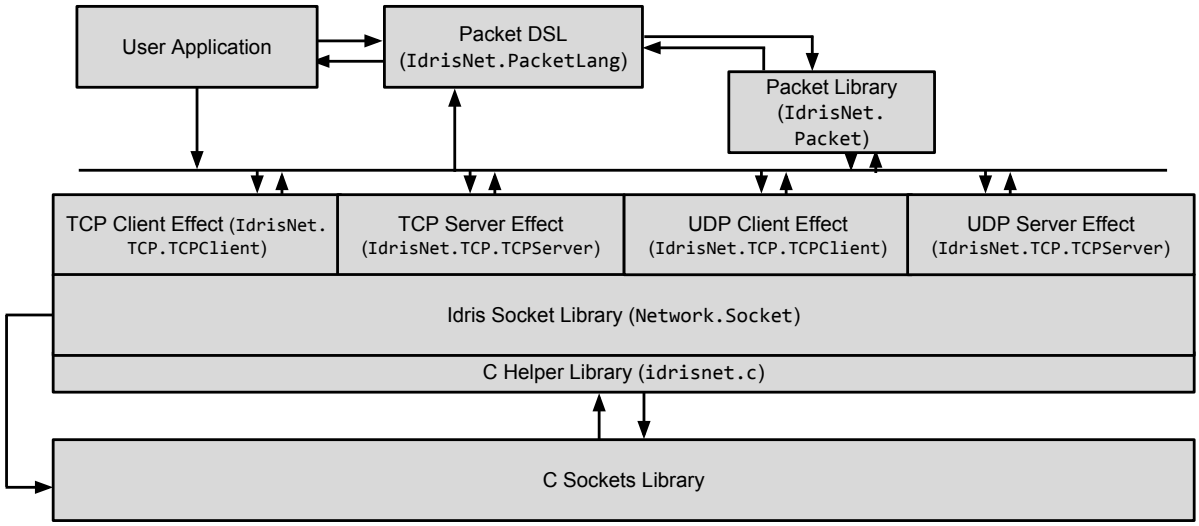


Figure 7.1: A high-level overview of the library

7.2 Network Library

7.2.1 Socket Library

The `IdrisNet.Socket` library provides a relatively low-level interface to the C socket API.

Initial prototypes of the library [1] did not provide access to this socket interface, instead providing coarser-grained granularity to support protocol-specific operations. Additionally, much of the socket setup routines were provided in the unmanaged C-code level, using a C-based structure to maintain state across operations.

[1] http://www.github.com/SimonJF/IdrisNet

An example of this would be listening for connections on a TCP server socket. Instead of providing the C socket operations at the Idris level, a TCP server file instead appealed to a routine which created, bound, and listened on a socket at the C level.

This approach was abandoned for a variety of reasons: firstly, the granularity of the operations was much too coarse, and therefore resulted in a lack of flexibility. An example of this would be a user wishing to bind to a socket, but defer listening on the socket to a later point in the execution of a program. Secondly, since multiple operations took place in each unmanaged function call, error reporting became more difficult, as it was necessary to return which particular operation caused an error should the entire computation fail. Thirdly, the C sockets API is very general, allowing for many different protocols, socket types and addressing modes to be used. In the original model, these were not exposed to the user, resulting in a lack of flexibility.

The approach used within the final version of the project involves a low-level Idris API binding to the C socket library, which makes calls directly to the C socket libraries via the foreign function interface (FFI) where possible. Where this is not possible, for example where additional code such as setting up a 'hints' structure when binding to a local address, small helper functions in a much thinner wrapper library (`idrisnet.c`) are called.

7.2.1.1 API

The library implements the API functions specified in Figure 7.2, which will be familiar to the reader with a background in network programming.

```
socket : SocketFamily ->
         SocketType ->
         ProtocolNumber ->
         IO (Either SocketError Socket)

close : Socket -> IO ()
bind : Socket -> (Maybe SocketAddress) -> Port -> IO Int
connect : Socket -> SocketAddress -> Port -> IO Int
listen : Socket -> IO Int
send : Socket -> String -> IO (Either SocketError ByteLength)
recv : Socket -> Int -> IO (Either SocketError (String, ByteLength))
sendBuf : Socket -> Ptr -> Int -> IO (Either SocketError ByteLength)
recvBuf : Socket -> Ptr -> Int -> IO (Either SocketError ByteLength)
accept : Socket -> IO (Either SocketError (Socket, SocketAddress))
sendTo : Socket -> SocketAddress -> Port ->
  String -> IO (Either SocketError ByteLength)
sendToBuf : Socket -> SocketAddress -> Port ->
  BufPtr -> ByteLength -> IO (Either SocketError ByteLength)
recvFrom : Socket -> ByteLength ->
  IO (Either SocketError (UDPAddrInfo, String, ByteLength))
recvFromBuf : Socket -> BufPtr -> ByteLength ->
  IO (Either SocketError (UDPAddrInfo, ByteLength))
accept : Socket -> IO (Either SocketError (Socket, SocketAddress))
```

Figure 7.2: The User-Facing API for `IdrisNet.Socket`

These API functions simply expose the lower-level functionality in Idris, providing similar type

signatures to those of the C APIs. Several data types and type synonyms are used, however, to ensure readability and add additional type safety guarantees.

```
record Socket : Type where
  MkSocket : (descriptor : SocketDescriptor) ->
             (family : SocketFamily) ->
             (socketType : SocketType) ->
             (protocolNumber : ProtocolNumber) ->
             Socket

data SocketFamily = AF_UNSPEC -- Unspecified
                  | AF_INET   -- IP / UDP etc. IPv4
                  | AF_INET6  -- IP / UDP etc. IPv6

ProtocolNumber : Type
ProtocolNumber = Int

SocketError : Type
SocketError = Int

SocketDescriptor : Type
SocketDescriptor = Int

data SocketAddress = IPv4Addr Int Int Int Int
                   | Hostname String
```

Figure 7.3: The Socket Type

Figure 7.3 describes the data types used to represent a socket in Idris. In particular, the native file descriptor referring to the socket is stored as an integer, and each socket is associated with a particular addressing family (for example, IPv4 or IPv6), a type of socket (for example, TCP or UDP), and an optional protocol number.

Underlying calls to the C socket library generally either return 0 if the socket connection has been closed by the remote host (for streaming sockets), -1 if an error has occurred (setting the `errno` value), or another positive value (for example, the number of bytes sent or received) if the operation has succeeded. We encapsulate the result of such operations using a sum type, `Either`, which is an algebraic data type parameterised over two type variables `a` and `b`, detailing that an inhabitant of the type may either be of type `a` or `b`.

7.2.2 Protocol Bindings

With the lower-level socket functionality exposed in the `IdrisNet.Socket` library, we may make use of the `Effects` framework to implement higher-level, dependently-typed bindings for protocols such as TCP.

7.2.3 TCP

TCP is implemented on top of the C Sockets library simply by specifying a socket type of `SOCK_STREAM` and an address family of `AF_INET` for IPv4, or `AF_INET6` for IPv6. In our implementation, we use

this to implement TCP functionality purely through the `IdrisNet.Socket` library, without needing to implement additional TCP-specific functionality in the C helper library.

At the same time, correctly making use of TCP sockets involves using a distinct resource usage protocol for both clients and servers, which we model through the use of dependent algebraic effects.

Sockets may be used either as TCP servers or TCP clients, and the resource usage protocols and permitted operations differ for each usage. For example, servers must firstly bind to an address, listen for clients, and accept and handle incoming clients. For this reason, we use two separate effects, `TCPCLIENT` and `TCPSERVER` instead of one, coarser-grained `TCP` effect.

Socket operations such as `bind`, `listen`, `send` and `recv` may either succeed, fail with a fatal error, fail with a recoverable error (such as `EAGAIN` or `EWOULDBLOCK`), or indicate that the remote host has closed the connection. In order to encapsulate this within the client and server effects, we declare an ADT, `SocketOperationRes` parameterised over the type of an operation should it succeed.

```
data SocketOperationRes a = OperationSuccess a
                          | FatalError SocketError
                          | RecoverableError SocketError
                          | ConnectionClosed
```

7.2.3.1 TCP Client Effect

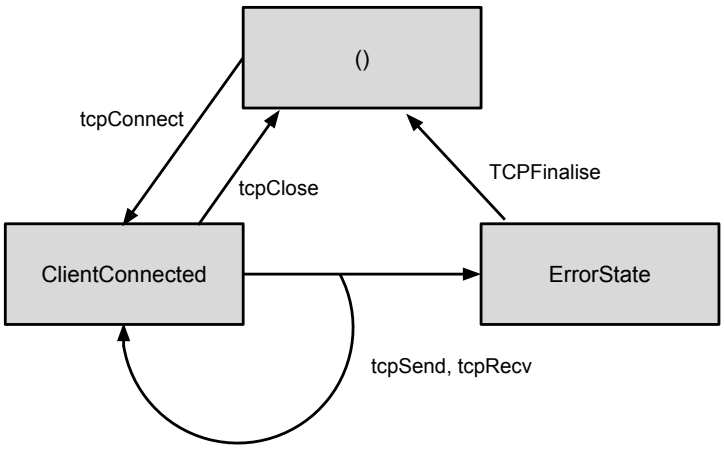


Figure 7.4: State transition diagram for the TCP Client effect

Figure 7.4 shows the state transition diagram for the TCP client effect. Execution begins in the uninitialised state, denoted by the unit resource `()`. By connecting to a remote host using the `tcpConnect` function, we transition into the `ClientConnected` state, where it is possible to read and write data to the stream, and close the connection.

Reading and writing may fail, however, and this is handled by the `Effects` library by interpreting the result of the operation and possibly transitioning into a state denoting failure, `ErrorState`. In order to encode these transitions, we define a function `interpOperationRes` which takes a `SocketOperationRes`, and returns the type of the output resource.

```
interpOperationRes : SocketOperationRes a -> Type
interpOperationRes (OperationSuccess _) = ClientConnected
interpOperationRes (FatalError _) = ErrorState
interpOperationRes (RecoverableError _) = ClientConnected
```

```
interpOperationRes ConnectionClosed = ()
```

We also define a function `interpConnectRes` which returns `ClientConnected` if successful, and `()` otherwise.

With this in place, we may define the abstract TCPClient effect, shown in Figure 7.5.

```
data TCPClient : Effect where
  Connect      : SocketAddress ->
                 Port ->
                 {() ==> interpConnectRes result}
                  TCPClient (SocketOperationRes Socket)

  Close        : { ClientConnected ==> ()} TCPClient ()

  Finalise     : { ErrorState ==> () } TCPClient ()

  WriteString  : String ->
                 { ClientConnected ==> interpOperationRes result}
                  TCPClient (SocketOperationRes ByteLength)

  ReadString   : ByteLength ->
                 { ClientConnected ==> interpOperationRes result }
                 TCPClient (SocketOperationRes (String, ByteLength))

  WritePacket  : (pl : PacketLang) ->
                 (mkTy pl) ->
                 { ClientConnected ==> interpOperationRes result }
                 TCPClient (SocketOperationRes ByteLength)

  ReadPacket   : (pl : PacketLang) ->
                 Length ->
                 { ClientConnected ==> interpOperationRes result }
                 TCPClient (SocketOperationRes (Maybe (mkTy pl, ByteLength)))
```

Figure 7.5: TCP Client Effect

The client effect details seven operations:

Connect
Attempts to establish a connection with a remote host on the given address and port. If successful, then transitions into the `ClientConnected` state. If not, then remains in the uninitialised state, denoted by the unit type `()`.

Close
Closes a currently open socket, and transitions back into the uninitialised state.

Finalise
Closes a socket in an erroneous state.

WriteString
Writes a string to the currently open socket. Requires the socket to be in the `ClientConnected` state, and based on the result of the operation, either remains in the ClientConnected state if

the operation was successful, or transitions to the ErrorState state if not. If successful, returns the number of bytes written to the stream.

ReadString
Reads a string from the socket, with the same input and output transitions as above. If successful, returns a tuple of the string that was read from the socket and the length of the string read from the socket.

WritePacket
Uses the packet language DSL described in Section 7.3, taking in a packet description and a corresponding concrete implementation and writing this to the socket. The input and output transitions are as above. If the operation was successful, then the length of the written data is returned.

ReadPacket
Similarly, uses the packet language DSL to attempt to unmarshal data given a particular packet description.

From this model, we may derive several correctness guarantees:

Theorem 1. *It is not possible to write to, or read from, a socket without first establishing a connection.*

Proof. Send and receive operations require the effect to be in the `ClientConnected` state. It is only possible to transition into the `ClientConnected` state after a successful `Connect` operation, which establishes a connection. Failure to do so, or check that the operation succeeded, will result in a compile-time type error.

Theorem 2. *It is not possible to write to, or read from, a socket if the connection has been invalidated by the failure of a previous operation.*

Proof. As above, send and receive operations require the effect to be in the `ClientConnected` state, and the failure of an operation will result in a transition to the `ErrorState` state. Attempting to perform a read or write operation whilst in the `ErrorState` state, or without checking if a read or write operation has succeeded, will result in a compile-time type error.

Theorem 3. *By specifying a program with uninitialised input and output resource types, it is not possible to write a total program which does not close any open handles.*

Proof. If no operations are performed, no handles are created. If the `Connect` operation is attempted but fails, no connection will be made, and the resource will remain uninitialised. If a connection is successfully created, the effect will transition into the `ClientConnected` state. In order to return the effect into the uninitialised state, either the `Close` or `Finalise` operations must be called, which close the connection. Failure to call these will result in a compile-time type error.

7.2.3.2 TCP Server Effect

Figure 7.6 shows the state transition diagram for the TCP server effect. Server sockets operate in a different way to client sockets, in that they must firstly bind to a port, listen for incoming connections, and then accept and work with the incoming collections.

In order to encapsulate this within the effect, we define five different states: the unit type `()` once again represents an uninitialised socket, `ServerBound` represents a socket that has been bound to an address and a port, `ServerListening` indicates that a socket is listening for connections, and `ErrorState` indicates that an error has occurred and the socket is therefore unusable. Both the

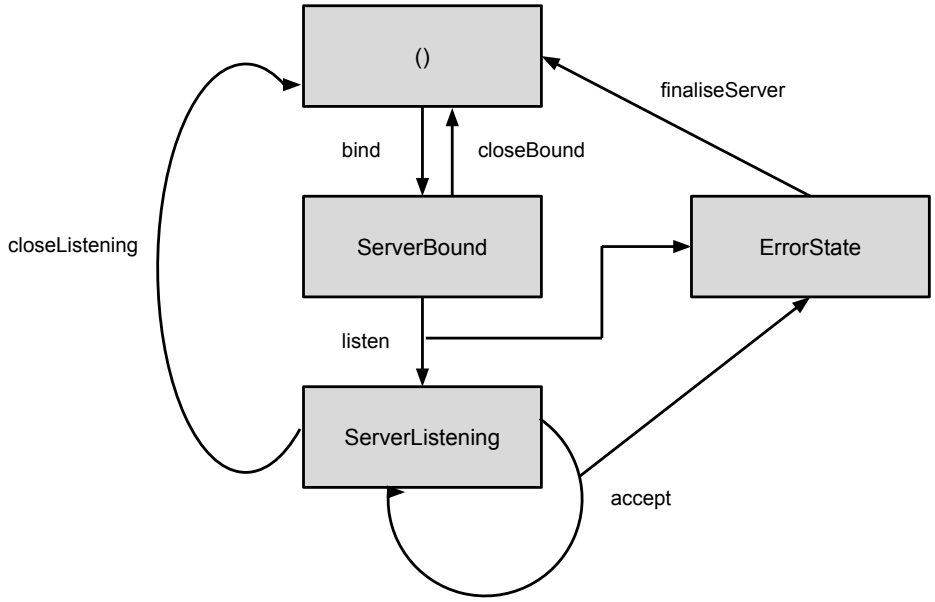


Figure 7.6: state transition diagram for the TCP server effect

`Listen` and `Accept` operations may fail, therefore transitioning into the `ErrorState` state, but it is not necessary to define separate failure states for each as the cleanup operation is identical.

Clients are accepted using the `Accept` and `ForkAccept` operations. The difference between these two operations is that `Accept` accepts the client in the same thread, meaning that no further clients will be accepted until the program handling the newly-accepted program has ceased execution. The `ForkAccept` operation, on the other hand, spawns a new VM thread to handle the new client.

In order to accept a client, a function of type `ClientProgram` must be specified, which is a type synonym as described below.

```
ClientProgram : Type -> Type
ClientProgram t = {[TCPSERVERCLIENT (ClientConnected)] ==>
                   [TCPSERVERCLIENT ()]} Eff IO t
```

The `TCPSERVERCLIENT` effect is identical to that of the `TCPCLIENT` effect, but does not include functionality for connecting to remote clients. The `ClientProgram` type specifies that the effectful program passed to `Accept` begins in the `ClientConnected` state, allowing read and write operations to be performed on the newly-accepted client, and ends in the uninitialised state, indicating that the accepted socket must be closed.

Our model of a TCP server socket allows us to derive further correctness guarantees.

Theorem 4. *It is not possible to attempt to accept clients without firstly successfully binding to, and listening on, a socket and port.*

Proof. The required input resource for the `Accept` and `ForkAccept` operations is `ServerListening`. In order to transition to the `ServerListening` state, it is firstly necessary to call the `Bind` operation, which transitions into the `ServerBound` state and binds to the socket, and secondly necessary to call the `Listen` operation, which listens on the bound socket. Failure to perform these two steps and check that they have completed successfully will result in compile-time type error.

Theorem 5. *It is not possible to perform operations on a socket that has been invalidated by the previous operation.*

Proof. The input resource for the `Bind` operation is `ServerListening`. If the `Listen` operation fails, then the effect will transition into the `ErrorState` state, and thus not fulfil this requirement. The required input resource for the `Accept` and `ForkAccept` operations is `ServerBound`. If the the `Bind` operation fails, then the effect will transition into the `ErrorState` state, thereby not fulfilling this requirement. If a call to `Accept` or `ForkAccept` fails, the effect will transition into the `ErrorState` state.

Theorem 6. *Assuming a total `ClientProgram` function, all resources used by a new client that has been accepted using the `Accept` or `ForkAccept` operations will be released.*

Proof. The type of `ClientProgram` specifies an input resource of `ClientConnected` and an output resource of `()`. If the program passes the totality checker—that is, it terminates without looping infinitely—the `Close` function must be called in order to transition to the `()` state, thereby releasing the resources.

7.2.4 UDP

UDP is similarly implemented using the underlying `IdrisNet.Socket` library, making use of the DGRAM socket type and the asynchronous and connectionless `sendto` and `recvfrom` functions.

In spite of being a simpler protocol, UDP servers have a resource usage protocol: if `recvfrom` is to be used, the socket must firstly be bound to a port. Although `sendto` and `recvfrom` make no guarantees about the arrival of data when used with datagram sockets, they will return a code denoting success should the operation be attempted successfully, regardless of whether the data arrived successfully. An error occurring at this point indicates a failure with the bound socket, meaning that no further operations should be attempted.

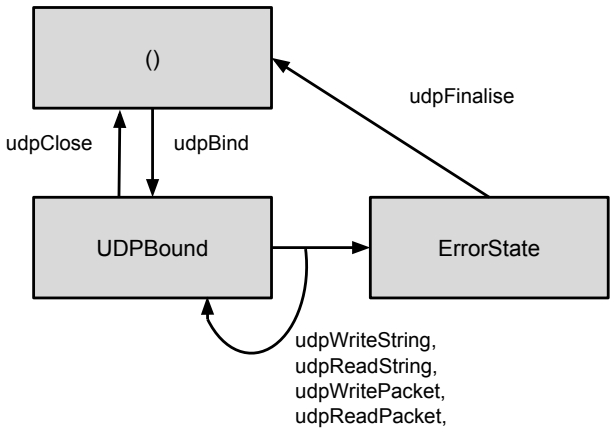


Figure 7.7: State transition diagram for the UDP server effect

The operations on the `UDPSERVER` effect are similar to those of the `TCPCLIENT` effect, along with the ability to bind to a port. In order to send and receive datagrams, programs making use of the `UDPSERVER` effect must firstly successfully bind to an address and port. Since UDP is a connectionless protocol, send operations also require a `SocketAddress` and `Port`, and receive operations also additionally return a data structure of type `UDPAddrInfo`, which contains the `SocketAddress` and `Port` of the remote client.

Similarly to the `TCPClient` and `TCPServer` effects, the results of effectful operations are encapsulated in an ADT parameterised over the result type, allowing the resulting resource to be calculated from the function result.

Theorem 7. *It is not possible to attempt to receive data on an unbound socket.*

Proof. The `UDPSReceiveString` and `UDPSReadPacket` operations require an input resource of `UDPBound`. In order to transition from the uninitialised state to the `UDPBound` state, it is necessary to perform the `UDPSBind` operation. This binds the socket. Failure to do so, or failure to check that the operation was successful, results in a compile-time type error.

Theorem 8. *It is not possible to attempt to send or receive data on a socket that has been invalidated by a failed previous operation.*

Proof. All send and receive operations require an input resource of `UDPBound`, and either remain in the `UDPBound` state if the operation was successful, or transition to the `UDPError` state if the previous operation failed. If an operation fails and the effect therefore is in the `UDPError` state, then no further operations may be performed.

Theorem 9. *By specifying a program using the `UDPServer` effect with uninitialised input and output resource types, it is not possible to write a total program which does not close any open handles.*

Proof. If no operations are performed, no handles are created, and the resource remains uninitialised. If the `UDPSBind` operation is performed unsuccessfully, any intermediate handles are closed within the operation, and the resource remains uninitialised. If the `UDPSBind` operation is completed successfully, the state will transition to `UDPBound`, and the `Close` operation, which closes all opened resources, must be called to transition back into the uninitialised state. If a read or write operation is unsuccessful and triggers a transition into the `ErrorState` state, the `UDPSFinalise` operation must be called in order to transition to the uninitialised state, which closes all open resources. If a program is written which specifies an uninitialised output resource, but one of the code paths results in a non-unit output resource, this will result in a compile-time type error.

In addition to the stateful implementation of UDP servers described above, we also include a second effect, `UDPCLIENT`, which may be used to send UDP packets without needing to be bound to a socket.

7.3 Packet DSL

The `PacketLang` DSL facilitates the declarative specification of packets, and allows for low-level data to be marshalled and unmarshalled. `PacketLang` was first exposited by Brady [7] as a demonstration of how Idris could be used for low-level systems programming.

As Idris is very actively developed, however, the original implementation in the paper is outdated and no longer compiles. Additionally, new advances in the language (for example, dependent algebraic effects) allow us to use the DSL in different ways to statically enforce additional invariants. For these reasons, we present a rewritten version of `PacketLang` to take advantage of the new language features, and show how they may be used to implement a verified version of a nontrivial protocol type, DNS.

The core DSL implementation is contained within `PacketLang.idr`, which specifies all of the core constructs and required syntax rewrite rules. The `Packet.idr` file contains the logic to marshal and unmarshal high-level data to its binary representation.

A helper library (`bindata.c`) is used for bit-level manipulation, although this remains unmodified from the original version by Edwin Brady.

7.3.1 PacketLang

The `PacketLang` DSL consists of a number of constructs. The language itself is specified by induction-recursion [8], allowing functions on the DSL to be defined at the same time as the DSL itself. In this case, the `mkTy` function is used to represent each language construct as a concrete Idris type.

By specifying the DSL in this manner, we may create two functions: one to describe the *structure* of the packet, and one to describe the *data* contained in the packet. The data contained in the packet, since it makes use of the structure in its type, must conform to the packet specification. A mismatch between the specification and implementation will result in a compile-time type error.

For example, we may define a simple packet description `simplePacket` consisting of two null-terminated strings.

```
simplePacket : PacketLang
simplePacket = do
  cstring
  cstring
```

We may then define an instance of this packet using the `mkTy` function, which translates the description into Idris types using the `simplePacket` description we have already defined. This relies on the use of type-level computation: we construct the Idris type for the implementation, and then may type check our implementation against this.

```
simplePacketInstance : (mkTy simplePacket)
simplePacketInstance = "Hello" ## "World!"
```

```
mutual
  data PacketLang : Type where
    CHUNK : (c : Chunk) -> PacketLang
    IF : (test : Bool) ->
         (yes : PacketLang) ->
         (no : PacketLang) ->
         PacketLang
    (//) : PacketLang -> PacketLang -> PacketLang
    LIST : PacketLang -> PacketLang
    LISTN : (n : Nat) -> PacketLang -> PacketLang
    NULL : PacketLang
    (>>=) : (p : PacketLang) -> (mkTy p -> PacketLang) -> PacketLang

  -- Packet language decoding
  mkTy : PacketLang -> Type
  mkTy (CHUNK c) = chunkTy c
  mkTy (IF x t e) = if x then (mkTy t) else (mkTy e)
  mkTy (l // r) = Either (mkTy l) (mkTy r)
  mkTy (LIST x) = List (mkTy x)
  mkTy (LISTN n a) = Vect n (mkTy a)
  mkTy NULL = ()
  mkTy (c >>= k) = (x ** mkTy (k x))
```

Figure 7.8: Constructs in the `PacketLang` DSL

Figure 7.8 details the core `PacketLang` language constructs. These are described in more detail below.

IF
: The `IF` construct takes a Boolean test parameter, along with two `PacketLang` descriptions. If the Boolean test evaluates to true, then the program will attempt to unmarshal the next part of data according to the first `PacketLang` argument, but will attempt to unmarshal using the second `PacketLang` description should the test evaluate to false.

`(//)`
: The `(//)` construct denotes choice, and corresponds to the sum type `Either`. The constructor takes two arguments, which are possible packet descriptions.

`LIST`
: The `LIST` construct specifies to a list of arbitrary length, and corresponds to the Idris `List` type. The construct takes a `PacketLang` parameter, which specifies the type of the list.

`LISTN`
: The `LISTN` construct is a *bounded* list, containing `n` values, where `n` is a natural number specified in the constructor.

NULL
: The `NULL` construct indicates that nothing should be parsed and no input consumed or written. It corresponds to the unit type `()`.

(>>=)
: The `(>>=)` construct denotes sequencing within the language. Since Idris makes use of ad-hoc syntax overloading, overloading the bind function allows us to make use of `do`-notation to specify packets. Since some packet specifications may depend on the form of data earlier in the specification, standard monadic binding syntax allows for fields in the specification to be given names, and used in later constructions.

  The `(>>=)` construct takes in two arguments: a `PacketLang` specification `p`, and a continuation function `k` making use of the decoded value of `p`. The Idris type corresponding to the binding construct is a dependent pair or Σ-type `((x ** mkTy (k x))`. This allows the types of later values to depend on values specified earlier in the packet.

  In order to avoid the need to pattern match on multiple nested dependent pairs, we define a syntax macro `(x ## y)` which corresponds to the dependent pair `(x ** y)`, but reassigns the associativity in such a was as to avoid the need to add extra parentheses.

The `CHUNK` construct specifies a chunk of binary data. The `Chunk` data type, as shown in Figure 7.9, specifies several different types of binary data. These abstract specifications are mapped to concrete Idris data types through the use of the `chunkTy` function.

`Bit` The `Bit` data type represents an positive integer represented using `width` bits. Constructing an instance of this requires a proof that the bit width is nonzero, but for statically-determinable bit sizes, we may make use of the possibility to specify a default proof to attempt to satisfy this obligation.

The `so` type is defined as follows:

```
data so : Bool -> Type where
  oh : so True
```

```
data Chunk : Type where
  Bit : (width : Nat) -> so (width > 0) -> Chunk
  CBool : Chunk
  CString : Chunk
  LString : Int -> Chunk
  Prop : (P : Proposition) -> Chunk


chunkTy : Chunk -> Type
chunkTy (Bit w p) = Bounded w
chunkTy CString = String
chunkTy (LString i) = String
chunkTy (Prop p) = propTy p
chunkTy (CBool) = Bool
```

Figure 7.9: The `Chunk` data type

As there is no constructor which may generate an instance of type `so False`, it is uninhabited. It follows, therefore, that if the predicate specified in the type is false, then the type is uninhabited and no proof exists.

```
bit : (w : Nat) -> {default tactics { refine oh; solve; }
                         p : so (w > 0) }
                 -> Chunk
bit w {p} = Bit w p
```

In this case, the `bit` function takes a `Nat` specifying the bit width, and attempts to discharge the proof obligation by applying the `oh` constructor. If the Boolean condition specified in the type holds, then the proof obligation will be automatically discharged. If the condition can be statically determined to be false, or there is not enough static information, then this will result in a compile-time type error.

The corresponding Idris type is `Bounded`, which is defined as follows:

```
data Bounded : Nat -> Type where
  BInt : (x : Int) ->
         (prf : so (x < (pow 2 i))) ->
         Bounded i
```

The `Bounded` type is parameterised over a `Nat`, which specifies the maximum number of bits required to hold a number of that type. For example, if we have a type of `Bounded 5`, we may store a value which may be contained in 5 bits (between 0 and 31).

The `Bounded` type is declared as follows:

```
data Bounded : Nat -> Type where
  BInt : (x : Int) ->
         (prf : so (x < (pow 2 i))) ->
         Bounded i
```

Since `i` is unbound in the type and data constructors, it is implicitly bound to the `Nat` in the type constructor, referring to the maximum number of bits bounding the value.

In order to construct a value of type `Bounded i`, we must also supply a proof that the given value will fit in that number of bits. If this is statically determinable, the proof obligation may be discharged as above. If not, which is more often the case, we may make use of the `choose` function defined in the Prelude, which gives takes a Boolean as its argument, returning either a proof that it is true or a proof that it is not true.

`CBool`
The `CBool` data type represents a Boolean value, and is stored as a 1-bit flag within the raw binary data. The corresponding Idris type is Bool. As would be expected, if the flag is set to 1, then the unmarshalled Boolean value is `True` and `False` if not.

This data type was not present in the original version of `PacketLang` [7], as flags could be represented as 1-bit bounded integers. Since Boolean flags are heavily used within packets, however, they have been included as a data type in their own right for ease of use.

`CString`
The `CString` data type represents a string of characters of variable length, terminated by a null character in the binary data. The corresponding Idris type is `String`.

`LString`
The `LString` data type represents a string of characters of fixed length. It is important to note that this length is *not* written to the packet as it can be determined from the specification. The corresponding Idris type is again `String`.

`Decodable`
The `Decodable` data type represents an integer within a packet description which may be decoded into a particular data type.

```
Decodable : (n : Nat) ->
            (t : Type) ->
            (Bounded n -> Maybe t) ->
            (t -> Bounded n) -> Chunk
```

The first parameter specifies the number of bits to store the encoded value. The second parameter specifies the type of the decoded value. The third parameter is a decoding function, which specifies a partial mapping from the bounded number to its data type representation. The fourth parameter is a total mapping from the data type to its bounded counterpart.

The corresponding Idris type is the type specified by the `t` parameter.

`Prop`
The `Prop` data type represents a proposition about the data. This is not written to the packet, but exists to ensure that data conforms to certain constraints.

Propositions about the data are specified by the `Proposition` data type.

```
data Proposition : Type where
  P_EQ : DecEq a => a -> a -> Proposition
  P_BOOL : Bool -> Proposition
  P_AND : Proposition -> Proposition -> Proposition
  P_OR : Proposition -> Proposition -> Proposition


-- Decode propositions into Idris types.
```

```
propTy : Proposition -> Type
propTy (P_EQ x y) = x=y
propTy (P_BOOL b) = so b
propTy (P_AND s t) = Both s t
propTy (P_OR s t) = Either (propTy s) (propTy t)
```

P_EQ

The P_EQ proposition states that two given values are decidably equal: that is, as well as determining that two values are equal, we may also provide a *proof* that this is the case.

In order to do so, we make use of the DecEq type class. Instances of this type class must implement the decEq function:

```
class DecEq t where
  total decEq : (x1 : t) ->
                (x2 : t) ->
                Dec (x1 = x2)
```

The decEq function tests for equality, returning either a proof that the two values are equal, or a proof that they are not. This proof may then either be used by the user to satisfy a proof obligation imposed by the proposition, or by the unmarshalling code to prove that the data satisfies the proposition.

P_BOOL

The P_BOOL proposition states that a given Boolean value is true. This is again done using the so type, using the oh data constructor if the condition holds.

P_AND

The P_AND proposition states that two other propositions hold.

The implementation of this has been rewritten since the original PacketLang implementation, and corresponds to an Idris type Both.

```
data Both : Proposition -> Proposition -> Type where
    MkBoth : (propTy a) ->
             (propTy b) ->
             Both a b
```

The Both data type is parameterised over two propositions a and b. In order to create a value of type Both a b, it is necessary to specify proofs that both propositions hold.

P_OR

The P_OR proposition states that either one of two propositions holds. This naturally corresponds to the sum type Either, and in order to create a value of type Either a b, it is necessary to specify either a proof of a using the Left data constructor, or a proof of b using the Right data constructor.

7.3.2 PacketLang Syntax

In order to make writing PacketLang specifications more user-friendly, we make use of the syntax rewriting rules provided by Idris. This results in the grammar of valid PacketLang expressions shown in Figure 7.10.

Expressions using these syntax rules are translated through two translation functions $\mathcal{S}$ (Figure 7.11), which translates core PacketLang constructs and binary chunks, and $\mathcal{P}$ (Figure 7.3.2), which translates propositions.

⟨*pl*⟩ ::= do ⟨*ds*⟩

⟨*ds*⟩ ::= ⟨*string*⟩ ← ⟨*item*⟩ ⟨*ds*⟩ | ⟨*item*⟩ ⟨*ds*⟩ | ⟨*item*⟩

⟨*item*⟩ ::= bits ⟨*nat*⟩
| check ⟨*bool*⟩
| lstring ⟨*nat*⟩
| cstring
| listn ⟨*nat*⟩ ⟨*item*⟩
| list ⟨*item*⟩
| p_if ⟨*item*⟩ 'then' ⟨*item*⟩ 'else' ⟨*item*⟩
| p_either ⟨*item*⟩ ⟨*item*⟩
| bool
| decodable ⟨*nat*⟩ ⟨*type*⟩ ⟨*function*⟩ ⟨*function*⟩
| prop ⟨*prop*⟩

⟨*prop*⟩ ::= prop_bool ⟨*bool*⟩
| prop_or (prop ⟨*prop*⟩) (prop ⟨*prop*⟩)
| prop_and (prop ⟨*prop*⟩) (prop ⟨*prop*⟩)
| prop_eq ⟨*eq*⟩ ⟨*eq*⟩

Figure 7.10: Syntax for the `PacketLang` DSL

$$
\begin{array}{lll}
\mathcal{S}[\![\ \texttt{bits}\ n\ ]\!] & \mapsto \texttt{CHUNK (bit n)} & (\text{if } n > 0) \\
\mathcal{S}[\![\ \texttt{check}\ b\ ]\!] & \mapsto \texttt{CHUNK (bit n)} & \\
\mathcal{S}[\![\ \texttt{cstring}\ ]\!] & \mapsto \texttt{CHUNK (CString)} & \\
\mathcal{S}[\![\ \texttt{lstring}\ n\ ]\!] & \mapsto \texttt{CHUNK (LString n)} & \\
\mathcal{S}[\![\ \texttt{bool}\ ]\!] & \mapsto \texttt{CHUNK (CBool)} & \\
\mathcal{S}[\![\ \texttt{null}\ ]\!] & \mapsto \texttt{CHUNK (NULL)} & \\
\mathcal{S}[\![\ \texttt{decodable}\ n\ ty\ fn1\ fn2\ ]\!] & \mapsto \texttt{CHUNK ((Decodable n ty fn1 fn2))} & \\
\mathcal{S}[\![\ \texttt{listn}\ n\ t\ ]\!] & \mapsto \texttt{LISTN n t} & \\
\mathcal{S}[\![\ \texttt{list}\ t\ ]\!] & \mapsto \texttt{LIST t} & \\
\mathcal{S}[\![\ \texttt{p\_if}\ p\ t\ e\ ]\!] & \mapsto \texttt{IF p t e} & \\
\mathcal{S}[\![\ \texttt{p\_either}\ p1\ p2\ ]\!] & \mapsto \texttt{(p1 // p2)} & \\
\mathcal{S}[\![\ \texttt{prop}\ p\ ]\!] & \mapsto \texttt{CHUNK (Prop } \mathcal{P}[\![p]\!]\texttt{)} &
\end{array}
$$

Figure 7.11: Syntax rewrite rules for the core `PacketLang` DSL

$$
\begin{array}{ll}
\mathcal{P}[\![\ \texttt{prop\_bool b}\ ]\!] & \mapsto \texttt{(P\_BOOL b)} \\
\mathcal{P}[\![\ \texttt{prop\_or (prop p1) (prop p2)}\ ]\!] & \mapsto \texttt{(P\_OR } \mathcal{P}[\![p1]\!]\ \mathcal{P}[\![p2]\!]\texttt{)} \\
\mathcal{P}[\![\ \texttt{prop\_and (prop p1) (prop p2)}\ ]\!] & \mapsto \texttt{(P\_AND } \mathcal{P}[\![p1]\!]\ \mathcal{P}[\![p2]\!]\texttt{)} \\
\mathcal{P}[\![\ \texttt{prop\_eq e1 e2}\ ]\!] & \mapsto \texttt{(P\_EQ p1 p2)}
\end{array}
$$

Figure 7.12: Syntax rewrite rules for propositions within `PacketLang`

7.4 Process Effect

Idris supports message-passing concurrency using the existing `System.Concurrency.Raw` (using untyped messages) and `System.Concurrency.Process` (using channels parameterised over a type). This works with the help of runtime system support: new VM instances may be spawned, and messages are sent between them by copying memory regions.

A problem with this in its existing form, however, is that it is incompatible with the `Effects` framework, as it works directly on top of the `IO` execution context. By implementing the module as an effect, it allows concurrency within effectful programs.

The `Process` effect is parameterised over a message type, which is used to communicate between threads. Handles to processes are represented as raw pointers: since these are inherently unsafe, they are typically wrapped within a data type. We specify a type `ProcPID`, parameterised by the message type.

```
data ProcPID msg = MkPPid Ptr
```

We then define a resource, `Running`, again parameterised over the message type.

```
data Running : Type -> Type where
  MkProc : Running mty
```

Spawned threads are designed to be used with the `Effects` framework, and as such we define a type synonym `RunningProcessM` which specifies the message type, execution context, input and output effects of the channel.

```
RunningProcessM : (mTy : Type) ->
                  (m : Type -> Type) ->
                  List EFFECT ->
                  List EFFECT ->
                  Type
RunningProcessM mty m effs effs' =
  Eff m () ((PROCESS (Running mty)) :: effs)
           (\_ => (PROCESS (Running mty)) :: effs')
```

The effect itself consists of operations for sending and receiving messages, checking whether messages have been received, and getting the local process ID. These are implemented by making calls to the functions within `System.Concurrency.Raw`, which in turn make calls to the runtime system.

Spawning a new thread involves specifying a message type, a function of type `RunningProcessM`, and an initial environment for the effects used within the thread. The handler function then spawns a new thread using the `fork` function, using the `runInit` function to run the effect.

```
handle MkProc (Spawn ty proc env) k = do
  ptr <- fork (runInit (MkProc :: env) proc)
  k (MkPPid ptr) MkProc
```

```
data Process : Effect where
  Spawn : (mty : Type) ->
          RunningProcessM mty IO effs effs' ->
          Env IO effs ->
          { (Running mty') } Process (ProcPID mty)

  -- Returns true if there's a message waiting in this process' mailbox,
  -- false if not
  HasMessageWaiting : { (Running mty) } Process Bool
  -- Sends a message to a given process
  SendMessage : ProcPID mty -> mty -> { (Running mty) } Process ()
  -- Receives a message from a given process
  RecvMessage : { (Running mty) } Process mty
  RecvMessageAddr : { (Running mty) } Process (ProcPID mty, mty)
  -- Gets local PID
  GetID : { (Running mty) } Process (ProcPID mty)
```

Figure 7.13: The Process Effect

Chapter 8

Implementation

In Chapter 7, we detailed the design of the network library and its components. In this chapter, we detail the implementation: the data structures, design patterns and algorithms used, the implementation challenges and their solutions, and examine example applications.

8.1 Socket Library

As described in Section 7.2.1, the socket library is implemented as a thin wrapper over the C sockets library in order to maximise flexibility for developers whilst still providing high-level, functional bindings. In order to implement this, we must have a method of executing native code: this is provided by the Idris foreign function interface (FFI).

8.1.1 Socket creation

The C sockets API is a standard abstraction over network functionality. A *socket* in this case is a handle which may be used in subsequent network operations. It is firstly necessary to initialise a socket by using the `socket` function call:

```
int socket(int domain, int type, int protocol);
```

The `domain` parameter denotes the address family of the socket. In practice, this corresponds as to whether the socket should be an IPv4 or IPv6 socket. In some circumstances, this may be left unspecified.

The `type` parameter specifies the *type* of socket to create. In practice, this specifies whether to create a stream (TCP) socket, or a datagram (UDP) socket. The `protocol` parameter specifies the protocol to be used. Generally this can simply be set to zero.

This functionality is exposed within Idris through the following function declaration:

```
socket : SocketFamily ->
         SocketType ->
         ProtocolNumber ->
         IO (Either SocketError Socket)
```

In this code `SocketFamily` and `SocketType` are ADTs which specify the address family and socket type respectively. `ProtocolNumber` is a type synonym for `Int`.

The return type of this is either a `SocketError`, which is a type synonym for `Int`, or a `Socket` record.

Later operations may need to make use of the parameters used when constructing this socket, and as such these are stored alongside the socket descriptor in the `Socket` record. This is defined as follows:

```
record Socket : Type where
  MkSocket : (descriptor : SocketDescriptor) ->
             (family : SocketFamily) ->
             (socketType : SocketType) ->
             (protocolNumber : ProtocolNumber) ->
             Socket
```

This also proveds a level of encapsulation over the `SocketDescriptor` field, which is simply a type synonym for `Int`.

Since the function call does not require any complex arguments or special setup, meaning that we may simply translate the ADT representations of the address family and socket type into their corresponding integer codes, and make a foreign function call:

```
socket : SocketFamily ->
         SocketType ->
         ProtocolNumber ->
         IO (Either SocketError Socket)
socket sf st pn = do
  socket_res <- mkForeign (FFun "socket" [FInt, FInt, FInt] FInt)
                  (toCode sf) (toCode st) pn
  if socket_res == -1 then -- error
    map Left getErrno
  else
    return $ Right (MkSocket socket_res sf st pn)
```

A common approach to reporting errors in C is to return -1, and set a variable called `errno` with an integer detailing the error that has occurred. In order to check whether an error occurred, we must therefore check whether or not the result of a socket function is -1, and if so, retrieve the value of `errno`.

This is simply implemented by writing an accessor function in the C helper library, and making a foreign function call.

```
int idrnet_errno() {
    return errno;
}

getErrno : IO Int
getErrno = mkForeign (FFun "idrnet_errno" [] FInt)
```

If successful, the return value of the `socket` function call will refer to the descriptor of the newly-created socket, which may then be used to construct a `Socket` instance.

8.1.2 Socket Binding

In order to listen for connections (in the case of TCP sockets) or data (UDP sockets), it is necessary to first bind to an address and port.

The C socket function `bind` is defined as follows:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

The function requires a socket descriptor, a `struct sockaddr` pointer detailing the address and port to which the socket should be bound, and an integer detailing the length of the given `struct sockaddr`. Since the Idris FFI does not provide an easy method to marshal a C structure, this functionality is delegated to the C helper library.

The helper function `idrnet_bind` requires a socket descriptor, the address family, socket type, a string detailing the remote host, and a port. Firstly, a `struct sockaddr` is populated through a call to `getaddrinfo()`. If this succeeds, then the socket is bound using the populated structure. If either stage of this computation fails, then -1 is returned and `errno` is set by the failing function. It is also important to free the `struct sockaddr` that has been allocated at this point.

8.1.3 Listening on a Socket

To listen on a socket, we make use of the `listen()` function, which is defined as follows:

```
int listen(int sockfd, int backlog);
```

The `listen()` function requires a socket descriptor and a `backlog` parameter, which defines the maximum number of queued connections to accept prior to refusing additional connections.

Since both parameters are primitives, we may call the function directly, without the need to use the C helper library. The Idris wrapper function simply makes the call, retrieving an error number if necessary.

8.1.4 Accepting Clients

A listening stream socket may accept sockets through the use of the `accept()` function.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The `accept()` function requires the socket descriptor of the listening socket, a pointer to a `struct sockaddr` to populate, and a pointer to a `socklen_t` to which the length of the populated structure will be written.

The type of the Idris `accept` binding is:

```
accept : Socket -> IO (Either SocketError (Socket, SocketAddress))
```

In order to implement this, it is necessary to make multiple FFI calls, since we need both the foreign socket descriptor and the remote socket address.

Accepting a client therefore involves firstly allocating enough memory to store the address information about the remote host, which is achieved though a call to `idrnet_create_sockaddr()`. This is then used as an argument to the `idrnet_accept()` function.

```
int idrnet_accept(int sockfd, void* sockaddr);
```

This function takes as its arguments the socket descriptor of the listening socket, and a pointer to the memory we previously allocated. The C helper library then passes these arguments to the underlying `accept` function, which returns either a new socket descriptor for the newly-accepted client, or -1 if an error has occurred.

We then check whether an error occurred, freeing all allocated memory and returning the error value if so. If not, then the final step is to unmarshal the socket address from the populated `struct sockaddr`.

In order to do this, we use two further helper functions, `idrnet_sockaddr_family()` to retrieve the address family, and `idrnet_sockaddr_ipv4()` to retrieve a string representation of an IPv4 address. This may then be parsed into a `SocketAddress` structure.

8.1.5 Sending Data

8.1.5.1 Stream Sockets

Sending data with stream sockets assumes that a connection has already been set up between two hosts. For this reason, it is not necessary to specify the remote host in the `send` call.

```
int send(int sockfd, const void *msg, int len, int flags);
```

As parameters, the `send` function takes a socket descriptor, a pointer to a buffer containing the data to send, the length of the buffer, and an integer containing any flags.

The `send` binding defined within the `IdrisNet.Socket` library takes a `Socket` record, and a string to send over the network. This suffices for simple types of packets, especially those based on textual data such as HTTP.

In addition to the `send` binding, we also provide a second function, `sendBuf`, which allows users to specify a pointer to the buffer to be sent. We make use of this when implementing the `PacketLang` DSL, the implementation of which is discussed in Section 8.4.

8.1.5.2 Datagram Sockets

Datagram sockets such as those used with UDP behave differently, as they do not require a connection to have been established prior to data being sent. Sending data using datagram sockets is instead done through the use of the `sendTo` function, which additionally requires some additional parameters specifying the destination address.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

In order to send data using this function, it is firstly necessary to create a `struct sockaddr` structure. The Idris binding to this function is defined as follows:

```
sendTo : Socket ->
     SocketAddress ->
         Port ->
         String -> IO (Either SocketError ByteLength)
```

To help marshal arguments from Idris structures to the required C arguments, we define a function `idrnet_sendto` in the C helper library.

```
int idrnet_sendto(int sockfd, char* data,
   char* host, int port, int family)
```

The file descriptor, string data and port may be used directly in the FFI call, and it is trivial to output a textual representation of the `SocketAddress` and `SocketFamily` arguments. The C helper library then uses these arguments to populate a `struct sockaddr` structure through a call to `getaddrinfo`, which may then be passed to the `sendto` function.

The `sendToBuf` function allows developers to specify a pointer to the data they wish to send, as with the `sendBuf` function.

8.1.6 Receiving Data

8.1.6.1 Stream Sockets

Receiving data is done through the use of the `recv` function.

```
int recv(int sockfd, void *buf, int len, int flags);
```

The `recv` function takes as its arguments a socket descriptor, a pointer to a buffer into which the received data should be stored, the length of the buffer, and an integer detailing any flags. If successful, the function will return the number of bytes read into the buffer. If the remote socket has been closed by the remote host, then calls to `recv` will return 0.

In order to both retrieve the received data and ascertain the result of the operation, we define the structure `idrnet_recv_result`, which stores the result and a pointer to the received data, and the helper operation `idrnet_recv`.

```
typedef struct idrnet_recv_result {
    int result;
    void* payload;
} idrnet_recv_result;

void* idrnet_recv(int sockfd, int len);
```

The `idrnet_recv` function returns a pointer to a `idrnet_recv_result` struct. The result of the call and the retrieved data may be accessed using the `idrnet_get_recv_res` and `idrnet_get_recv_payload` functions. After the required data has been extracted from the structure, it is freed using the `idrnet_free_recv_struct` function.

8.1.6.2 Datagram Sockets

As with sending data, datagram sockets do not require a pre-established connection to receive data. Instead, datagram sockets must *bind* to a socket and through a call to the `recvfrom` function, receive data and populate a `struct sockaddr` detailing the address information of the remote host from which the data was received.

As would be expected, the `recvfrom` function in C takes the same arguments as the `recv` function, but also requires a pointer to a `struct sockaddr` to be populated, along with a pointer to an integer into which the length of the populated structure will be stored.

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

At the Idris level, we wish to return either an error if any operation failed, or a 3-tuple detailing the address information of the remote host, the received data, and the length of the received data.

```
recvFrom : Socket ->
           ByteLength ->
           IO (Either SocketError
                      (UDPAddrInfo, String, ByteLength))
```

To implement this functionality, we firstly define a structure to hold the result of the `recvfrom` operation, the received data, and a pointer to a `struct sockaddr` detailing the address of the remote host.

```
typedef struct idrnet_recvfrom_result {
    int result;
    void* payload;
    struct sockaddr_storage* remote_addr;
} idrnet_recvfrom_result;
```

We then define a function in the C helper library, `idrnet_recvfrom()`, which takes in the socket descriptor of a bound datagram socket and the length of data to be received, which returns a pointer to a `idrnet_recvfrom_result` structure.

```
void* idrnet_recvfrom(int sockfd, int len)
```

Internally, this function allocates memory for `struct sockaddr` and `idrnet_recvfrom_result` structures and clears the allocated memory. It also allocates a buffer into which the retrieved data will be stored, clears this, and performs the underlying call to `recvfrom()`. The buffer is also null-terminated so that it may be safely sent over the FFI. The result, populated `struct sockaddr` structure, and pointer to the data buffer are all stored in the allocated `idrnet_recvfrom_result` structure and a pointer to this is returned. These can then be retrieved using accessor functions.

8.2 TCP Library

The dependently-typed TCP bindings are implemented through the use of the `Effects` library. Implementing an effect using this approach can be broken down into three steps: writing an abstract effect, which specifies the operations, their pre- and postconditions, and return types; writing handler functions which handle the abstract effect within an *execution context*; and finally writing wrapper functions which effectively "promote" an abstract effectual operation into a concrete operation which may be used within effectual programs.

In this section, we detail the implementation of the `TCPClient` and `TCPServer` effects. Whereas Section 7.2.3 details the *design* of the effects: that is, the state transitions and the safety guarantees that are given, this section instead concentrates on how these are implemented, paying specific attention to the handler functions.

8.2.1 Client

The `TCPClient` effect involves three possible states: uninitialised (signified by the unit resource `()`); `ClientConnected`, signifying that a connection has been successfully established to a remote host; and `ErrorState`, denoting that a fatal error has occurred during a socket operation and therefore the socket may not be used for further operations. These are concretely implemented as two ADTs, with constructors containing the `Socket` record.

```
data ClientConnected : Type where
  CC : Socket -> ClientConnected

data ErrorState : Type where
  ES : Socket -> ErrorState
```

The `Socket` record, detailing the socket used in the current connection, is stored in these resources so that it may later be used by effectual operations. It is important to note that this is completely hidden from the developer: at no point will they have to manually specify the socket to be used. The socket is also retained by the `ErrorState` state, so that the resources may be released by the `Finalise` operation.

After specifying the abstract `TCPClient` effect as in Section 7.2.3.1, it is necessary to promote this to a *concrete* effect: that is, one which may be used in effectual operations and may optionally carry a resource. This is achieved by defining a type constructor `TCPCLIENT` which is of kind `Type -> EFFECT`, where `EFFECT` is a type with a constructor `MkEff`, and `Type` is the type of a resource. `MkEff` takes as its arguments the type of a resource, and an abstract effect.

```
TCPCLIENT : Type -> EFFECT
TCPCLIENT t = MkEff t TCPClient
```

In order to handle failure, the `Effects` library allows state transitions to be predicated on the result of the previous operation. For example, for the `Connect` operation, we wish to remain in the

uninitialised state if the connection is unsuccessful, and transition into the `ClientConnected` state if the connection was successfully established. Let us revisit the definition of the `Connect` operation.

```
data TCPClient : Effect where
  Connect      : SocketAddress ->
                 Port -> {() ==> interpConnectRes result}
                         TCPClient (SocketOperationRes Socket)
```

Looking at each part of this operation in turn, we see that it firstly takes two arguments `SocketAddress` and `Port`, specifying the remote host to which a connection should be attempted. More importantly, we then see the type of the effect. Breaking this down, we firstly specify the pre- and post-conditions:

```
{() ==> interpConnectRes result}
```

This specifies that in order to perform this operation, we must firstly be in the uninitialised state. The result of the operation is bound to the variable `result`, which is used to calculate the state after the operation. We then specify the type of the effect, `TCPClient`, and the type of the result `(SocketOperationRes Socket)`.

The `interpConnectRes` may then use this to calculate the type of the output resource.

```
interpConnectRes : SocketOperationRes a -> Type
interpConnectRes (OperationSuccess _) = ClientConnected
interpConnectRes _ = ()
```

We use the same technique to encode the result of other socket operations which require a connection and may fail, through the `interpOperationRes` function.

```
interpOperationRes : SocketOperationRes a -> Type
interpOperationRes (OperationSuccess _) = ClientConnected
interpOperationRes (FatalError _) = ErrorState
interpOperationRes (RecoverableError _) = ClientConnected
interpOperationRes ConnectionClosed = ()
```

The next step is to write handler functions for each of the operations, which specify how the abstract effect is handled within the underlying execution context. Since we wish to make calls to the socket library, we must use an execution context which supports this, such as `IO`.

Most of the program logic is present in the `IdrisNet.Socket` module. The job of the effect handlers in this case is firstly to call the socket library with the correct parameters: for example, `AF_INET` and `SOCK_STREAM` for a TCP socket using IPv4, and secondly to interpret the results of the socket functions and perform the correct state transitions.

To specify handlers for an effect, we create an instance of the multi-parameter `Handler` type class, which is parameterised over the abstract effect and the execution context.

We look firstly at the handler for the `Connect` operation.

```
instance Handler TCPClient IO where
  handle () (Connect sa port) k = do
    sock_res <- socket AF_INET Stream 0
    case sock_res of
      Left err => k (FatalError err) ()
      Right sock => do
        conn_res <- connect sock sa port
        if (conn_res == 0) then -- Success
          k (OperationSuccess sock) (CC sock)
```

```
        else do
          close sock
          k (FatalError conn_res) ()
```

To handle an effect, we specify a function `handle`, taking the current resource, the operation, and a continuation function `k` as its arguments. Recalling the type of the `Connect` operation, we may pattern match to ascertain the `SocketAddress` parameter `sa` and the port, and use this in further calls. Since we are running within the `IO` context, we may make calls to the `Network.Socket` library within the handler function.

We firstly attempt to create a new socket, through the use of the `socket` function. If unsuccessful, then this will return an error value using the `Left` constructor. If this is the case, then we fail with a fatal error and specify an empty resource by passing these as arguments to the continuation function.

If the function call was successful, then a populated `Socket` record will be returned using the `Right` constructor. We then use this socket to try and connect to the remote host. If this is successful, then we return `OperationSuccess` and update the resource to `ClientConnected`, using the `CC` constructor. If not, then the socket is closed, `FatalError` is returned, and the resource remains uninitialised.

The remaining operations follow the same pattern: they make calls to the `Network.Socket` library, interpret the results, and update the resource as appropriate.

The final stage of implementing the effect involves defining simple wrapper functions around the operations, allowing them to be used in effectful programs. For example, we define the wrapper around the `Connect` operation:

```
tcpConnect : SocketAddress ->
             Port ->
             { [TCPCLIENT ()] ==> [TCPCLIENT (interpConnectRes result)] }
             Eff IO (SocketOperationRes Socket)
tcpConnect sa port = (Connect sa port)
```

The type declaration for the `tcpConnect` function is almost identical to that of the abstract `Connect` operation, except that it is of type `Eff`, and specifies the execution context and a list of *concrete* effects and their associated resources.

8.2.2 Server

The `TCPServer` effect has a slightly more complex resource usage protocol, in that a server socket may be uninitialised, bound, listening, or in an error state, but these are implemented in the same manner as the TCP client effect.

Of particular note is the method by which we handle new clients. Recall that when accepting a client, we must specify a function of type `ClientProgram`.

```
ClientProgram : Type -> Type
ClientProgram t = {[TCPSERVERCLIENT (ClientConnected)] ==>
                   [TCPSERVERCLIENT ()]} Eff IO t
```

A function of type `ClientProgram` operates on the socket opened by accepting a client, and must close all open resources when it finishes execution. In order to implement this functionality from within the handler for the `Accept` operation, we call the `runInit` function from within the handler function.

```
handle (SL sock) (Accept prog) k = do
  accept_res <- accept sock
  case accept_res of
```

```
Left err => do
  if err == EAGAIN then
    k (RecoverableError err) (SL sock)
  else
    k (FatalError err) (SE sock)
Right (client_sock, addr) => do
  res <- runInit [(CC client_sock addr)] prog
  k (OperationSuccess res) (SL sock)
```

The `runInit` function takes an initial environment, and runs an effectful program. In this case, we initialise the environment with the client socket and remote address, and run the given program. After this function has finished executing, the remainder of the server program executes.

8.3 UDP Library

Both the UDP server and client effects are implemented in exactly the same fashion as above, but instead initialise the socket with the `Datagram` socket type, and make use of the `sendto` and `recvfrom` functions instead of `send` and `recv`.

8.4 Packet DSL

As described in Section 7.3, the core DSL constructs are implemented through the use of induction-recursion in order to simultaneously define the data type and functions on the data type.

The `mkTy` function translates between DSL constructs and their corresponding Idris types. We similarly use type-level computation in the form of the `chunkTy` function translate between `Chunk` declarations and their corresponding types, and `propTy` to translate propositions.

8.4.1 Calculating Packet Length

Calculating the length of a packet is achieved using the `bitLength` function, which takes as its arguments the `PacketLang` description and its corresponding data.

```
bitLength (CHUNK c) x = chunkLength c x
bitLength (IF True yes _) x = bitLength yes x
bitLength (IF False _ no) x = bitLength no x
bitLength (y // z) x = either (\l_x => bitLength y l_x)
                              (\r_x => bitLength z r_x) x
bitLength (LIST pl) x = listLength pl x
bitLength (LISTN n pl) x = vectLength pl x
bitLength NULL _ = 0
bitLength (c >>= k) (a ** b) = bitLength c a + bitLength (k a) b
```

To calculate the length of a `IF` construct, we match on the Boolean predicate: if this holds, then we calculate the length of the argument based on the first packet description, whereas if it does not, we calculate the length based on the second description.

To calculate the length of the `//` construct, we match on the constructor of the data, `x` using the `either` function to get the packet specification, which is then used to calculate the length of the data.

Calculating the length of a `LIST` and `LISTN` constructs involves recursively calculating the sum of the lengths of each element, using the `pl` description.

The binding construct $(>>=)$ involves sequencing multiple constructs together. The binding construct involves a packet language description `c` and a continuation function `k`. Recall that the

continuation function takes the *value* associated with `c` as its argument, allowing it to be used in further computations within the packet specification. The data associated with a binding construct is a dependent pair consisting of the data corresponding to `c`, `a`; and the remaining data in the packet, `b`. We firstly calculate the length of `a`, and secondly calculate the length of `b` by providing `a` as an argument to the continuation function.

To calculate the length of a binary chunk, we make use of the `chunkLength` function.

```
chunkLength : (c : Chunk) -> chunkTy c -> Length
chunkLength (Bit w p) _ = natToInt w
chunkLength CBool _ = 1
chunkLength CString str = 8 * ((strLen str) + 1)
chunkLength (LString len) str = 8 * len
chunkLength (Decodable n _ _ _) _ = natToInt n
chunkLength (Prop _) p = 0
```

The length of a fixed-width number is the width. This is stored in the `Bit` chunk as a natural number, and is converted using the `natToInt` function. The same applies for a decodable type, which is stored as a fixed-width number. The length of a Boolean flag is one bit. The length of a fixed-width string is the number of bytes required to hold the string, multiplied by 8 to get the length in bits. The length of a null-terminated string is the length of the string, plus an extra character for the null terminator, multiplied again by 8. As propositions are not written to the packet, they have a length of zero.

8.4.2 Marshalling Data

The code to marshal and unmarshal data is contained within the `IdrisNet.Packet` library. The first step is to create FFI calls to the `bindata` C helper library, which allows us to create a packet by allocating a region of memory; set a byte at a given position; set a series of bits at a given position; and write a string.

```
foreignCreatePacket : Int -> IO BufPtr
foreignSetByte : BufPtr -> Position -> ByteData -> IO ()
foreignSetBits : BufPtr -> Position -> Position -> ByteData -> IO ()
foreignSetString : BufPtr -> Position -> String -> Int -> Char -> IO ()
```

We next define a top-level function `marshal`, which takes a `PacketLang` description and a corresponding implementation, returning a tuple of type `IO (BufPtr, Length)`, where the result of type `BufPtr` is a pointer (wrapped to add a layer of type-safety) to the raw packet in memory, and `Length` is the length of the written packet.

```
marshal : (pl : PacketLang) -> (mkTy pl) -> IO (BufPtr, Length)
```

This function firstly uses `bitLength` to calculate the required buffer size, makes the FFI call to create the packet buffer in memory, and calls the internal `marshal'` function.

The internal `marshal'` function requires an argument of type `ActivePacket`, which details the maximum length of the packet and the current position to which data should be written. The result is the length of data that was written: this may then be used to determine the position that the next piece of data should be written.

```
data ActivePacket : Type where
  ActivePacketRes : BufPtr -> BytePos -> Length -> ActivePacket

marshal' : ActivePacket -> (pl : PacketLang) -> mkTy pl -> IO Length
```

The rules for marshalling data according to `PacketLang` constructs are similar to those used to calculate the length of a packet. To marshal an `IF` construct, we check the predicate of the construct, marshalling according to the first given packet description if it holds and according to the second if it does not. Marshalling the `(//)` construct involves again matching on the type of data using the `either` function, marshalling according to the first packet description if the data was constructed with the `Left` constructor, and according to the second packet description if the data was constructed with the `Right` constructor.

Marshalling lists and vectors involves writing each item in turn, through the use of the `marshalList` and `marshalVect` functions.

Recall once more the form of the binding construct $(>>=)$, which takes a packet language description `c` and a continuation function `k`, with an associated data value `(a ** b)`, where `a` is the data corresponding to `c` and `b` is the remaining data in the packet.

```
marshal' ap (c >>= k) (a ** b) = do
  len <- marshal' ap c x
  let (ActivePacketRes pckt pos p_len) = ap
  let ap2 = (ActivePacketRes pckt (pos + len) p_len)
  len2 <- marshal' ap2 (k x) y
  return $ len + len2
```

To handle this, we firstly marshal `b` according to the `PacketLang` specification `a` and record the length. We then marshal the rest of the packet, applying `x` to the continuation function `k` to allow it to be used in the rest of the specification. Finally, we return the sum of the data lengths.

8.4.2.1 Marshalling Chunks

Marshalling actual chunks of binary data is achieved through the use of the `marshalChunk` function.

```
marshalChunk : ActivePacket -> (c : Chunk) -> (chunkTy c) -> IO Length
marshalChunk (ActivePacketRes pckt pos p_len) (Bit w p) (BInt dat p2) = do
  let len = chunkLength (Bit w p) (BInt dat p2)
  foreignSetBits pckt pos (pos + (natToInt w) - 1) dat
  return len
marshalChunk (ActivePacketRes pckt pos p_len) CBool b = do
  let bit = if b then 1 else 0
  foreignSetBits pckt pos pos bit
  return (chunkLength CBool b)
 ...
```

To marshal a bounded integer, we call the `foreignSetBits` function, which makes an FFI call to the underlying `setPacketBits` function. The function requires four arguments: the packet, the start index, the end index, and the bits to set (encoded as an integer). As a Boolean value is a special case of this, the same function is used. The same technique is used to marshal a `Decodable` chunk, after applying the encoding function.

Marshalling strings is achieved through a call to the `foreignSetString` function, which calls `setPacketString` in the C library. This function requires five arguments: the packet, the start index, the string, the length, and a termination character. The same technique may be used for both null-terminated and bounded strings.

8.4.3 Unmarshalling Data

To unmarshal data, we define a function `unmarshal` which takes as its arguments a packet language description, a pointer to a data buffer, and the length of the packet.

```
unmarshal : (pl : PacketLang) ->
            BufPtr ->
            Length ->
            IO (Maybe (mkTy pl, ByteLength))
```

Parsing may fail, which is encapsulated through the use of the `Maybe` type. We name the packet description `pl`, and use this in the return type. The function, as with marshalling, constructs an `ActivePacket` instance, and calls the inner `unmarshal'` function.

8.4.3.1 Unmarshalling `PacketLang` constructs

`IF`

To unmarshal according to the `IF` construct, we evaluate the Boolean predicate which will either be a constant or be determined by data that has been previously unmarshalled. If this evaluates to true, then we attempt to unmarshal the next section with the first given packet description. If it evaluates to false, then we attempt to unmarshal the next section with the second packet description.

`(//)`

To unmarshal according to the `(//)` construct, we firstly try and unmarshal the next section using the first given `PacketLang` description. If this succeeds, then we return the unmarshalled data wrapped in the `Left` constructor, indicating that the first description was used. If it fails, then we attempt to unmarshal with the second packet description. If this succeeds, we return the unmarshalled data wrapped in the `Right` constructor.

`LIST`

In order to unmarshal according to the `LIST` construct, we repeatedly attempt to unmarshal data using the given packet description until either we encounter data that fails to parse, or we reach the end of the packet. This functionality is implemented using the `unmarshalList` function:

```
unmarshalList : ActivePacket ->
                (pl : PacketLang) ->
                (List (mkTy pl), Length)
unmarshalList (ActivePacketRes pckt pos p_len) pl =
  case (unmarshal' (ActivePacketRes pckt pos p_len) pl) of
    Just (item, len) =>
      let (rest, rest_len) =
        unmarshalList (ActivePacketRes pckt (pos + len) p_len) pl in
          (item :: rest, len + rest_len)
    Nothing => return ([], 0)
```

We firstly attempt to unmarshal a piece of data using the packet description. If this succeeds, then we offset the location by the length of the parsed data, and parse the rest of the list. If not, then this is the base case and the empty list cell is returned, along with a length of zero. We then build the list recursively.

`LISTN`

Unmarshalling a list with a fixed length is simply achieved by attempting to unmarshal exactly the given number of data items, and putting these into a list. If parsing data fails at any point, then the `LISTN` parse fails and `Nothing` is returned.

(>>=)
Unmarshalling the binding operation works by firstly attempting to unmarshal data using the packet language description `c`. If this is successful, we attempt to unmarshal the remainder of the data through the use of the continuation function `k`, passing the unmarshalled value as an argument, thereby allowing it to be used in further computations. If both of these unmarshalling operations are successful, we return them in a dependent pair alongside the combined length. If either fail, the entire parse fails and we return `Nothing`.

```
unmarshal' (ActivePacketRes pckt pos p_len) (c >>= k) = do
  (res, res_len) <- unmarshal' (ActivePacketRes pckt pos p_len) c
  (res2, res2_len) <-
     unmarshal' (ActivePacketRes pckt (pos + res_len) p_len) (k res)
  return ((res ** res2), res_len + res2_len)
```

8.4.3.2 Unmarshalling Chunks

`Bit`

To unmarshal a bounded integer, we firstly check to see whether the requested range of bits is within the total length of the packet. If so, we call the `foreignGetBits` function, which is an FFI call to the `getPacketBits` function in the `bindata` library.

`CBool`

Unmarshalling a Boolean value is a specialisation of the above procedure: we retrieve the value of the bit at the current position, returning `True` if it is 1, and `False` otherwise.

`Decodable`

To unmarshal a `Decodable` value, we read in the integer using the `foreignGetBits` function, and decode using the specified decoding function. If this decoding function succeeds, then the value is extracted from the `Just` constructor and returned. If not, parsing fails.

`CString`

The process of unmarshalling a null-terminated string involves recursively unmarshalling a byte at a time, until a null byte is reached. We define two functions: one an outer function `unmarshalCString`, and one an inner function `unmarshalCString'`.

```
unmarshalCString : ActivePacket -> IO (Maybe (String, Length))

unmarshalCString' : ActivePacket ->
                    Int ->
                    IO (Maybe (List Char, Length))
```

The inner `unmarshalCString'` function firstly checks to see whether the next character is within the bounds of the packet. If so, then it retrieves the byte using the `getPacketBits` function, translating the byte into an Idris character using the inbuilt `chr` function. This character is then prepended onto the rest of the string, which is recursively calculated. If a null character is encountered, this is treated as the end of the string and an empty list is returned.

The `unmarshalCString` function calls the inner function, and translates the list of characters to a more efficient `String` type.

`LString`

Unmarshalling a string with a pre-specified length works in a similar way, but instead of recursively retrieving each character until a null terminator is found, we retrieve the set number of characters after checking that the string is within the range of the packet.

8.4.3.3 Unmarshalling Propositions

P_EQ

Recall that the arguments of a P_EQ proposition must be members of the DecEq class, which allows us to check whether or not the arguments are *decidably equal*: that is, we may construct a proof of their equality.

To gain this proof (or ascertain that such a proof cannot be constructed), we use the decEq function included in the Idris prelude. This function returns a value of type Dec, which has two constructors: Yes, indicating that the proposition in the type must hold, and No, indicating that the proposition cannot hold. To construct a value of Yes, we must specify a proof that the proposition holds, whereas to construct a value of No, we must specify a contradiction.

```
class DecEq t where
  total decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)

data Dec : Type -> Type where
  Yes : {A : Type} -> (prf : A) -> Dec A
  No : {A : Type} -> (contra : A -> _|_) -> Dec A
```

By matching on the result of the decEq function, we may extract the equality proof if the result is a value constructed using the Yes constructor, whereas we return Nothing if the value is constructed using the No constructor.

P_BOOL

The P_BOOL proposition specifies that a given Boolean value must be true. This is translated into the so type, which we may only construct using oh if the proposition holds. To construct such a proof dynamically, we use the choose function from the prelude which returns either a proof that a Boolean is true, or a proof that it is not. If it is, then we match on Left to get the proof to return, returning Nothing if not.

P_AND

The P_AND proposition indicates that two given propositions must hold. To construct such a proof, we firstly recursively evaluate the results of both propositions, and if they both hold, then we may construct a value of the Both type by providing both proofs.

P_OR

The P_OR proposition indicates that at least one of two given propositions must hold. To construct this proof, we firstly evaluate the first proposition: if this holds, then we return this proof using the Left constructor. If it does not, we attempt to evaluate the second proposition. If this holds, then we return it using the Right constructor. If neither hold, then the P_OR proposition does not hold and we return Nothing.

8.5 Example Programs

In this section, we discuss how the different components of the system may be combined to create example programs. We begin with a simple example, a TCP client / server system where the client reads in input, sends it to the server, and waits for the server to repeat the data back.

We then examine two larger case studies: a PacketLang encoding of DNS packets, and a networked Pong game.

8.5.1 Echo Client / Server

8.5.1.1 Server

We firstly begin by defining a function, `setupServer`, which binds to, and listens on, a server socket.

```
setupServer : Maybe SocketAddress -> Port -> Bool ->
              { [TCPSERVER (), STDIO] } Eff IO ()
setupServer sa port do_fork = do
  putStr "Binding\n"
  OperationSuccess _ <- bind sa port
    | RecoverableError _ => return ()
    | FatalError err => do
        putStr ("Error binding: " ++ (show err) ++ "\n")
        return ()
    | ConnectionClosed => return ()
  putStr "Bound\n"
  OperationSuccess _ <- listen
    | RecoverableError err => do
        putStr ("Recoverable error: " ++ (show err))
        closeBound
    | FatalError err => do
        putStr ("Error binding: " ++ show err)
        finaliseServer
    | ConnectionClosed => return ()
  putStr "Listening\n"
  if do_fork then forkServerLoop else serverLoop
```

Note that in the type declaration, we make use of two effects: `TCPSERVER ()`, indicating that we wish to use the `TCPSERVER` effect, and `STDIO`, indicating that we wish to communicate with standard input and standard output. Since the associated resource with `TCPSERVER` is uninitialised both at the start and end of execution, this means that all resources that are opened along any execution path must also be closed.

In this function, we firstly attempt to bind to the given socket address and port by calling the `bind` operation. If this is successful (indicated by a return value of `OperationSuccess`), we then attempt to listen on the socket. If this also succeeds, then we proceed into the main server loop. Failures are handled by printing a message detailing the notion of the failure, and, if resources have been acquired by the point of failure, releasing the resources.

The next step is to define a client acceptance loop, `serverLoop`, which accepts clients as they arrive.

```
serverLoop : { [TCPSERVER (ServerListening), STDIO] ==>
               [TCPSERVER (), STDIO] } Eff IO ()
serverLoop = do
  OperationSuccess _ <- accept receive
    | RecoverableError _ => serverLoop
    | FatalError err => do putStr ("Error accepting: " ++ (show err))
                           finaliseServer
    | ConnectionClosed => return ()
  serverLoop
```

This function simply calls the `accept` operation using the `receive` function as the program to execute. Once again, if failures occur then any associated resources must be released, as specified by the type signature.

The `receive` function is merely a wrapper to introduce the `STDIO` effect. The main logic is then defined in the `receive'` function, which reads from the TCP socket, and if successful, writes the same string back. If an error occurs, or the connection is closed, then the client socket is closed.

```
receive' : { [STDIO, TCPSERVERCLIENT ClientConnected] ==>
             [STDIO, TCPSERVERCLIENT ()] } Eff IO ()
receive' = do
  -- Receive
  OperationSuccess (str, len) <- tcpRecv 1024
    | RecoverableError _ => receive'
    | FatalError err => do putStr ("Error receiving: " ++ (show err))
                           tcpFinalise
    | ConnectionClosed => return ()
  -- Echo
  OperationSuccess _ <- tcpSend str
    | RecoverableError err => do putStr ("Error sending: " ++ (show err))
                                 tcpClose
    | FatalError err => do putStr ("Error sending: " ++ (show err))
                           tcpFinalise
    | ConnectionClosed => return ()
  receive'
```

Finally, the main entry point of the application runs the `setupServer` effect with the required arguments.

```
main : IO ()
main = run (setupServer Nothing 1234 False)
```

8.5.2 Client

The client program is constructed in a similar manner: we firstly define a function `echoClient` to attempt a connection using the `tcpConnect` function. If this succeeds, then we call a second function, `clientLoop`, which retrieves input from a user, sends it to the server, and awaits and prints a the response.

The code is very similar to that of the server.

8.5.3 DNS

The DNS packet system is of interest for several reasons: there are several invariants within the packet that may be encoded using `PacketLang`; there are several points at which data in DNS packets depends on previous values; and the compression system presents interesting challenges in packet decoding.

While we implement some of the (backwards-compatible) changes implemented in subsequent revisions of DNS, we concentrate mainly on the standard as defined in RFC 1035 [28]. The ASCII diagrams that follow are taken from this RFC unless otherwise specified.

8.5.3.1 Packet Specification

DNS Packets

All DNS queries and responses make use of the same general packet structure, shown in Figure 8.1

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

Figure 8.1: High-Level Overview of DNS Packets

Each packet begins with a `Header` section which details metadata regarding the rest of the packet. Secondly, in the case of query packets, there may be one or more entries in the `Question` section, which detail the type and data of the requests to be sent to the server. The final three fields have a common format: that of a Resource Record (RR).

Header Section

```
                                    1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA| Z|AD|CD|   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 8.2: DNS Header Section

The header section shown in section 8.2 is as defined in RFC 2535 [9]: it supersedes that of the original RFC1035 implementation by adding two extra flags in place of a previously reserved space.

The header section begins with a 16-bit unique identifier for the DNS request. The `QR` bit is unset if the message is a query, and set if it is a response. The `OPCODE` field is a 4-bit number which details the sort of query (for example, standard or inverse). The `Z` bit must be set to zero, and the `RCODE` field specifies a response code detailing possible error codes.

Additionally the header consists of four 16-bit integers `QDCOUNT`, `ANCOUNT`, `NSCOUNT`, and `ARCOUNT` detailing the number of questions, answers, name servers and additional records respectively.

To encode the header and the outer DNS packet structure in `PacketLang`, we change the interpretation of the header somewhat, and separate the record counts into the outer structure. By doing this, we may more easily link the counts and the number of records to parse within the structure.

```
dns : PacketLang
dns = with PacketLang do
  header <- dnsHeader
  qdcount <- bits 16
  ancount <- bits 16
  nscount <- bits 16
  arcount <- bits 16
  questions <- listn (intToNat $ val qdcount) dnsQuestion
  answers <- listn (intToNat $ val ancount) dnsRR
  authorities <- listn (intToNat $ val nscount) dnsRR
  listn (intToNat $ val arcount) dnsRR
```

Figure 8.3: `PacketLang` representation of the outer DNS packet structure

Figure 8.3 shows the `PacketLang` specification of a DNS packet.

We begin by specifying a `dnsHeader` implementation, followed by the four 16-bit integers specifying the number of records in each of the sections. Since we may bind the values of these due to the use of induction-recursion within the DSL definition, we may therefore use these values in specifying the lengths of the fields. This ensures that the number of records within each section is the same as the number of records specified within the packet. We must use the `val` and `intToNat` functions to translate the bounded integer definitions into natural number representations for use within the `listn` constructor.

The identifier and each Boolean flag are coded using the `bits` and `bool` constructs. The DNS RFC demands that the reserved `Z` flag is zero: we encode this within the specification by inserting a Boolean proposition `check (not z)` which ensures that this is the case. This means that it is not possible to create a packet in which this invariant does not hold.

We also wish to ensure that only valid DNS opcodes and responses are used. It is also useful to unmarshal these to an Idris data type. We specify these invariants using the `decodable` construct, which takes as its arguments the length of the integer representation, the data type to which the encoded data should be unmarshalled, and decoding and encoding functions. Since there exists a total surjective mapping from data type constructors to bounded integer representations, it can be guaranteed that packets constructed using the given specification cannot specify an out-of-range value for the `OPCODE` or `RCODE` fields.

Question Section

Each DNS question section consists of three components: a variable-length section detailing a domain name, and two 16-bit integers detailing a question type and question class.

A DNS domain name is represented as a series of length-indexed strings, terminated by a null byte. Additionally, the DNS specification defines a compression scheme which may be used on a domain name, in which a pointer to a previous location in the packet may be used. More specifically,

```
dnsHeader : PacketLang
dnsHeader =
  with PacketLang do
     ident <- bits 16
     qr <- bool
     opcode <- decodable 4 DNSHdrOpcode
                 dnsCodeToOpcode dnsOpcodeToCode
     aa <- bool
     tc <- bool
     rd <- bool
     ra <- bool
     z  <- bool
     check (not z)
     ans_auth <- bool
     auth_acceptable <- bool
     decodable 4 DNSResponse
        dnsCodeToResponse dnsResponseToCode
```

Figure 8.4: `PacketLang` representation of a DNS Header

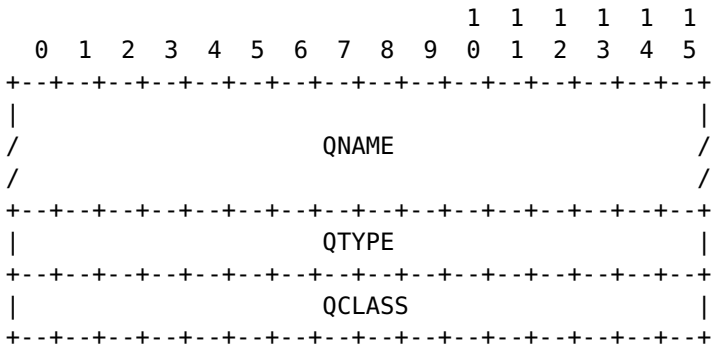


Figure 8.5: DNS Question Structure

a DNS domain field can be defined as either a reference, or a series of length-indexed strings terminated either by a null terminator or a reference. The distinction is made between a reference and a string length marker by setting the highest two bits.

Figure 8.6 shows how the DNS domain representation scheme may be represented in `PacketLang`. We define a function `tagCheck` which checks whether the two bits are set to a given value, returning the bits and two equality proofs if so. To decode a DNS label, we firstly check whether or not the first two bits are unset. If so, then the remaining 6 bits of the octet will determine the number of characters in the string segment. If this is null, then we have reached the end of the domain and stop parsing.

With this, the remainder of the DNS question section is easy to encode, as shown in Figure 8.5.3.1.

We specify that a `dnsQuestion` consists of a `dnsDomain` and two 16-bit integers that may be encoded and decoded into `DNSQType` and `DNSQClass` instances respectively. Once again this provides us with guarantees that no invalid types or classes may be encoded into the packet, and if a packet is

```
tagCheck : Int -> PacketLang
tagCheck i = do tag1 <- bits 1
                tag2 <- bits 1
                let v1 = val tag1
                let v2 = val tag2
                prop (prop_eq v1 i)
                prop (prop_eq v2 i)

dnsReference : PacketLang
dnsReference = do tagCheck 1
                  bits 14

dnsLabel = do tagCheck 0
              len <- bits 6
              let vl = (val len)
              prf <- check (vl /= 0)
              listn (intToNat vl) (bits 8)

dnsLabels : PacketLang
dnsLabels = do list dnsLabel
               nullterm // dnsReference

dnsDomain : PacketLang
dnsDomain = dnsReference // dnsLabels
```

Figure 8.6: `PacketLang` specification of DNS domains

```
dnsQuestion : PacketLang
dnsQuestion = do
  dnsDomain
  decodable 16 DNSQType dnsCodeToQType dnsQTypeToCode
  decodable 16 DNSQClass dnsCodeToQClass dnsQClassToCode
```

Figure 8.7: `PacketLang` specification of DNS Question segment

to successfully parse, these must be correct.

Resource Record Section

Resource records encode the data associated with a response. These may be direct answers to a query, data about a nameserver, or records containing additional information.

A resource record is specified in RFC1035 as shown in Figure 8.8.

The `NAME` field once again contains the DNS representation of a domain name as described in the above scheme. The `TYPE` and `CLASS` fields specify the type and class of the resource record, and therefore determine the form of the payload in the `RDATA` field. The `TTL` field is a 32-bit integer detailing the validity of the data, and the `RDLENGTH` field specifies the length of the payload.

Figure 8.9 details the `PacketLang` specification for DNS resource records. We firstly specify the DNS domain, and use the `decodable` construct to encode the `TYPE` and `CLASS` fields.

We then make use of an additional function, `dnsPayloadLang`, to calculate (based on the values

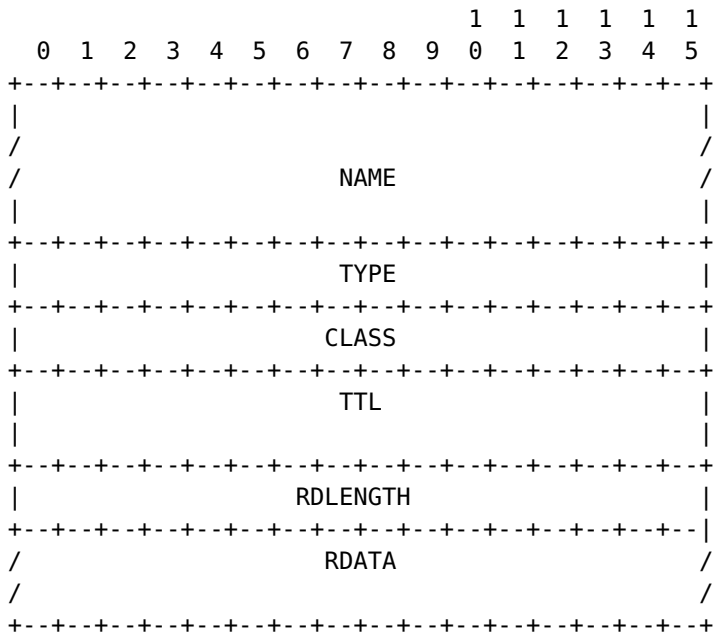


Figure 8.8: DNS Resource Record Structure

```
dnsRR : PacketLang
dnsRR = with PacketLang do
  domain <- dnsDomain
  ty <- decodable 16 DNSType dnsCodeToType' dnsTypeToCode'
  cls <- decodable 16 DNSClass dnsCodeToClass' dnsClassToCode'
  ttl <- bits 32
  len <- bits 16 -- Length in octets of next field
  let pl_lang = dnsPayloadLang ty cls
  pl_data <- pl_lang
  let data_len = (bitLength pl_lang pl_data) `div` 8
  prop (prop_eq (val len) data_len)
```

Figure 8.9: `PacketLang` specification for a DNS Resource Record

of the type and class) the specification to use to unmarshal the `RDATA` section. We also require a proof that the length of the encoded payload is equal to the value given in the `len` field.

```
dnsPayloadLang : DNSType -> DNSClass -> PacketLang
dnsPayloadLang DNSTypeA DNSClassIN = dnsIP
dnsPayloadLang DNSTypeAAAA DNSClassIN = null
dnsPayloadLang DNSTypeNS DNSClassIN = dnsDomain
dnsPayloadLang DNSTypeCNAME DNSClassIN = dnsDomain
dnsPayloadLang _ _ = null
```

Although we do not support the full range of DNS payloads, the `A` (IPv4 address), `SOA` (start of authority), `NS` (nameserver) and `CNAME` (canonical name) suffice to demonstrate the required concepts.

8.5.3.2 Packet Parser Implementation

Although the above `PacketLang` specifications precisely dictate the structure and invariants of packets, they remain at a fairly low level of abstraction: at this stage, if a packet successfully parses then we know that it is valid and that all proof obligations have been satisfied, but the proofs are likely of little use to users, and the data is not presented in a form in which it may be easily used. Finally, it remains necessary to decode the references in the packet to their corresponding expanded definitions.

As well as presenting the decoded data in a more accessible fashion, we do not wish for the representation of the data to be less precise.

We firstly define a data type `DNSPacket` to represent a top-level decoded DNS packet. This retains the correspondence between the count fields and the number of records in each section.

```
record DNSPacket : Type where
  MkDNS :
    (dnsPcktHeader : DNSHeader) ->
    (dnsPcktQDCount : Nat) ->
    (dnsPcktANCount : Nat) ->
    (dnsPcktNSCount : Nat) ->
    (dnsPcktARCount : Nat) ->
    (dnsPcktQuestions : Vect dnsPcktQDCount DNSQuestion) ->
    (dnsPcktAnswers : Vect dnsPcktANCount DNSRecord) ->
    (dnsPcktAuthorities : Vect dnsPcktNSCount DNSRecord) ->
    (dnsPcktAdditionals : Vect dnsPcktARCount DNSRecord) ->
    DNSPacket
```

We may then also define a data type, `DNSHeader`, to represent a DNS header. Note that there is no need to use dependent types in this section to represent the correspondence between the decoded integer values for opcodes and response codes and their higher level representations, as the decoding has been performed when unmarshalling the `Decodable` construct.

```
record DNSHeader : Type where
  MkDNSHeader :
    (dnsHdrId : Int) ->
    (dnsHdrIsQuery : Bool) ->
    (dnsHdrOpcode : DNSHdrOpcode) ->
    (dnsHdrIsAuthority : Bool) ->
    (dnsHdrIsTruncated : Bool) ->
```

```
    (dnsHdrRecursionDesired : Bool) ->
    (dnsHdrRecursionAvailable : Bool) ->
    (dnsAnswerAuthenticated : Bool) ->
    (dnsNonAuthAcceptable : Bool) ->
    (dnsHdrResponse : DNSResponse) ->
    DNSHeader
```

In a similar vein, it is trivial to represent the DNS question segment.

```
data DNSQuestion : Type where
  MkDNSQuestion :
    (dnsQNames : List DomainFragment) ->
    (dnsQQType : DNSQType) ->
    (dnsQQClass : DNSQClass) ->
    DNSQuestion
```

It is slightly more complex, however, to represent the resource records, as we must statically represent the dependency between the type and class fields and the payload: that is, it should not be possible to construct a `DNSRecord` instance with a payload that does not match the type and class fields.

```
data DNSRecord : Type where
  MkDNSRecord :
    (dnsRRName : List DomainFragment) ->
    (dnsRRType : DNSType) ->
    (dnsRRClass : DNSClass) ->
    (dnsRRTTL : Int) ->
    (dnsRRRel : DNSPayloadRel dnsRRType dnsRRClass pl_ty) ->
    (dnsRRPayload : DNSPayload pl_ty) ->
    DNSRecord
```

This invariant may be encoded using several design patterns often used when programming with dependent types. We firstly define a data type `DNSPayloadType`, which details the different types of payload.

```
data DNSPayloadType = DNSIPv4
                    | DNSIPv6
                    | DNSDomain
                    | DNSSOA
```

We then define a generalised algebraic data type parameterised over each payload type.

```
data DNSPayload : DNSPayloadType -> Type where
  DNSIPv4Payload : SocketAddress -> DNSPayload DNSIPv4
  DNSIPv6Payload : SocketAddress -> DNSPayload DNSIPv6
  DNSDomainPayload : List DomainFragment -> DNSPayload DNSDomain
  DNSSOAPayload : DNSSoA -> DNSPayload DNSSOA
```

Recall the type of the `dnsRRPayload` field in `DNSRecord`:

```
dnsRRPayload : DNSPayload pl_ty
```

In this case, `pl_ty` is a value of type `DNSPayloadType`, and its value is calculated from the `dnsRRType` and `dnsRRClass` fields.

We firstly calculate the payload type using the `payloadType` function:

```
payloadType : DNSType -> DNSClass -> Either DNSParseError DNSPayloadType
payloadType DNSTypeA DNSClassIN = Right DNSIPv4
payloadType DNSTypeAAAA DNSClassIN = Right DNSIPv6
payloadType DNSTypeNS DNSClassIN =  Right DNSDomain
payloadType DNSTypeCNAME DNSClassIN = Right DNSDomain
payloadType DNSTypeSOA DNSClassIN = Right DNSSOA
payloadType _ _ = Left PayloadUnimplemented
```

Once this is calculated, we must then find a way of linking the three values at the type level so that they may be used to calculate other types. In order to do this, we create a *relation* between the three types by specifying another GADT, `DNSPayloadRel`. This is parameterised over a type, class, and payload type, and the constructors indicate the allowed combinations.

```
data DNSPayloadRel : DNSType -> DNSClass -> DNSPayloadType -> Type where
  DNSPayloadRelIP : DNSPayloadRel DNSTypeA DNSClassIN DNSIPv4
  DNSPayloadRelIP6 : DNSPayloadRel DNSTypeAAAA DNSClassIN DNSIPv6
  DNSPayloadRelCNAME : DNSPayloadRel DNSTypeCNAME DNSClassIN DNSDomain
  DNSPayloadRelNS : DNSPayloadRel DNSTypeNS DNSClassIN DNSDomain
  DNSPayloadRelSOA : DNSPayloadRel DNSTypeSOA DNSClassIN DNSSOA
```

For example, the `DNSPayloadRelIP` relation may be used to state that a record of type `DNSTypeA` and class `DNSClassIN` will have a payload type of `DNSIPv4`.

By providing a *covering function*, we may specify a type, class, and payload type and from this retrieve a relation, should one exist.

```
getPayloadRel : (pl_ty : DNSPayloadType) ->
                (ty : DNSType) ->
                (cls : DNSClass) ->
                Either DNSParseError (DNSPayloadRel ty cls pl_ty)
getPayloadRel DNSIPv4 DNSTypeA DNSClassIN = Right DNSPayloadRelIP
getPayloadRel DNSIPv6 DNSTypeAAAA DNSClassIN = Right DNSPayloadRelIP6
getPayloadRel DNSDomain DNSTypeCNAME DNSClassIN = Right DNSPayloadRelCNAME
getPayloadRel DNSDomain DNSTypeNS DNSClassIN = Right DNSPayloadRelNS
getPayloadRel DNSSOA DNSTypeSOA DNSClassIN = Right DNSPayloadRelSOA
getPayloadRel _ _ _ = Left PayloadUnimplemented
```

By then providing the relation, we may specialise the `pl_ty` variable within the type, refining the `dnsRRPayload` field.

```
 ...
    (dnsRRRel : DNSPayloadRel dnsRRType dnsRRClass pl_ty) ->
    (dnsRRPayload : DNSPayload pl_ty) ->
 ...
```

Given this specialisation, only certain `DNSPayload` constructors may be used, meaning that we have additional guarantees about the nature of the *values* of the payload: it is not possible to construct a value of type `DNSPayload DNSDomain` using an IPv4 address, for example[1].

With the packet specifications and these higher-level data types defined, the process of translating between the two is relatively mechanical, so we omit most of the process here.

[1] The current implementation does not, however, use a parameterised GADT representation of socket addresses. This means that it is possible to create an instance of `DNSPayload IPv6` using an IPv4 address, for example: this is more a limitation of the socket library than the approach we use for DNS payloads, however.

Some techniques worth mentioning, however, are firstly the process of constructing a bounded integer: for example, in order to encode a header identifier, we must ensure that the identifier fits into 16 bits. To do this, we use the `isBounded` function, which tries to create a proof that the given value fits into the given number of bits, and if so, this is used in the `BInt` constructor. If not, then the operation fails with an encoding error.

```
isBounded : (bound : Nat) ->
            Nat ->
            Either DNSEncodeError (Bounded bound)
isBounded b n =
  case choose (i_n < (pow 2 b)) of
      Left yes => Right (BInt i_n yes)
      Right _ => Left $ OutOfBoundsError b n
  where i_n : Int
        i_n = natToInt n
```

The other important implementation detail is the decoding of references. References are extracted from the raw packet as bounded integers, and must be decoded into meaningful domains. This process is achieved by specifying an effect, `DNSParser`, which has a cache of previously-decoded references. When a reference is encountered, then the cache is firstly checked to see if it contains the reference and its associated decoded value. If so, then this is returned. If not, then the reference is parsed and the value cached.

This does, however, break some abstractions: in order to perform this, we must retain a pointer to the packet buffer, which is not optimal, and repeat lookups that we have already done. This is a limitation of the language as it stands, and we discuss this further in Section 9.

8.6 Putting it all together: Networked Pong

So far, we have seen how resource-dependent algebraic effects may be used to handle side effects within programs in a composable way; how effects may be used to enforce resource usage protocols in networked applications; how message-passing concurrency may be used within the `Effects` framework; and how an embedded domain-specific language may be used to marshal and unmarshal packet data.

This case study involves combining all of these components to create a larger application: a networked version of the popular Pong video game released by Atari in 1972. The game involves two players (or a player and a computer-controlled AI) each of whom control a paddle, hitting a ball back and forth.

To implement Pong, we require several different effects:

- `SDL`[2] to draw graphics and handle user input.
- `State` to maintain the game state, such as the position of the ball and paddles.
- `UDP Client` to make UDP requests to the other client.
- `UDP Server` to receive UDP packets from the other client.
- `Process` to spawn a thread to receive update packets.
- `StdIO` to print messages to the console.

[2] https://github.com/edwinb/SDL-idris

Additionally, we make use of `PacketLang` to specify the structure of the packets that are sent between the clients.

We begin by defining a type synonym, `Pong`, which details a program making use of these effects. Note that `UDPSERVER` is not included here, as it is used in the spawned thread as opposed to the main thread.

```
Pong : Type -> Type -> Type
Pong s t = { [ SDL s
               , STATE GameState
               , STDIO
               , UDPCLIENT
               , PROCESS (Running GameMessage)] } Eff IO t

PongRunning : Type -> Type
PongRunning t = Pong SDLSurface t
```

The type constructor takes two parameters: the resource type of the SDL effect, denoting whether the graphics library has been initialised or not, and the return type of the function. We additionally define another type synonym, `PongRunning`, which specifies that the SDL library has been initialised.

The architecture of the application involves one peer which acts as a server, and sets and transmits the motion of the ball. The position and velocity of the ball is transmitted every time it changes direction, and its position is then updated locally. Both peers communicate the motion of their respective paddles by sending a UDP packet upon the start and end of paddle motion.

We set up the UDP server by spawning a new thread, using the `GameMessage` data type to communicate between threads.

```
setupUDP : Port -> PongRunning ()
setupUDP p = with Effects do
  pid <- getPID
  spawn GameMessage (networkHandlerThread p pid) [(), ()]
  eventLoop
```

The `networkHandlerThread` function takes as its arguments the port on which to listen, and the process ID of the main thread. Once this is spawned, it attempts to bind to the given port and listen for status update messages. If messages are received and successfully unmarshalled, then they are decoded into a `GameMessage` using the `mkMessage` function, and communicated to the main thread.

```
networkHandlerThread : Port ->
                       (mthread : ProcPID GameMessage) ->
                       RunningProcess
                        GameMessage IO [UDPSERVER (), STDIO]
networkHandlerThread p pid = with Effects do
  putStr $ "Binding to port " ++ (show p) ++ "\n"
  UDPSuccess _ <- udpBind Nothing p
    | UDPFailure err => do putStr $ "Error binding: " ++ (show err)  ++ "\n"
                           return ()
    | UDPRecoverableError err => return ()
  networkHandlerThread' pid

networkHandlerThread' : (mthread : ProcPID GameMessage) ->
                        RunningProcessM GameMessage IO
                         [UDPSERVER UDPBound, STDIO]
```

```
                        [UDPSERVER (), STDIO]
networkHandlerThread' pid = with Effects do
  UDPSuccess (_, Just (pckt, _)) <- IdrisNet.UDP.UDPServer.udpReadPacket statusUpdate 256
    | UDPSuccess (_, Nothing) => do putStr "Error decoding status packet\n"
                                    networkHandlerThread' pid
    | UDPFailure err => do IdrisNet.UDP.UDPServer.udpFinalise
                           return ()
    | UDPRecoverableError err => do IdrisNet.UDP.UDPServer.udpClose
                                    return ()
  let msg = mkMessage pckt
  sendMessage pid msg
  networkHandlerThread' pid
```

The main event loop of the game is as follows.

```
eventLoop : PongRunning ()
eventLoop = do
  st <- get
  handleNetworkEvents
  continue <- handleKeyEvents !poll
  updateRemotePaddlePos
  paddle_updated <- updateLocalPaddlePos
  when paddle_updated (sendPaddleUpdate >>= \_ => return ())
  if (isServer st) then serverEvents else updateBallPos
  draw
  when continue eventLoop
```

Firstly, `handleNetworkEvents` is invoked, which checks whether there are any messages queued to be handled. If so, then the messages are handled by making any necessary changes to the state. Secondly, `poll` is used to check to see whether the user has entered any input, and if so, this is handled within the `handleKeyEvents` function which updates the game state. The paddles are then updated, based on whether the up or down keys are pressed for each client. If the up or down keys have been pressed, meaning that the paddle has changed direction, a packet is then sent to relay this information to the remote client. The ball position is then updated, either by performing the relevant collision checks and notifying the remote client of any changes, or by updating based on the current velocities of the ball.

These functions all use different subsets of the available effects, but this is all handled automatically, without any need to lift any operations.

Chapter 9

Evaluation

In this chapter, we evaluate the outcomes of the report to our original aims, and compare and contrast to existing related work.

As discussed in Chapter 2, the objectives specified in the initial objectives document served as possible avenues of exploration, and as such it soon became apparent that many of them were unachievable in the time available.

Nonetheless, I believe that all of the revised objectives have been met successfully, and in doing so, I believe the work has provided insights into how dependent types may be used in network applications. We now examine the revised objectives, and discuss how they have been addressed within this work.

Bindings to the C sockets library

This has been achieved by the `Network.Sockets` library. Using the Idris FFI and a C helper library, high-level bindings to the C sockets library have been created. These have overcome the limitations of the FFI only being able to return simple values at a time, and the limited set of data types which may be sent, and provide a sufficient level of abstraction to allow developers to work with this library in idiomatic code.

While most of the basic functionality has been implemented, some more advanced features such as setting socket options has not. As this is not the central focus of the project, we leave this to future work, should such functionality be required.

Verified bindings to TCP and UDP sockets

Using the `Network.Socket` library as a base, and making use of recent research on the `Effects` framework, this goal has been achieved by the `Network.TCP.TCPClient`, `Network.TCP.TCPServer`, `Network.UDP.UDPClient`, and `Network.UDP.UDPServer` modules. Achieving this goal was a result of analysing the possible failure cases of such sockets, and modelling these as states within an algebraic effect. As such, we have stated and proved several safety guarantees given by these modules, and shown how they guard against mistakes commonly made by application developers.

I believe that there are two main areas for improvement in this area. Firstly, in particular with the TCP effects, I believe that the operations should make more use of a stream-based abstraction instead of simply sending and receiving strings. Ideally, the user would not have to specify a buffer size parameter, and this would be handled as part of the abstraction. Implementation-wise, this would require keeping track of state within the effect, and reading strings terminated by newlines, for example.

The second area for improvement lies with the verbosity of having to handle possible failure cases. In an ideal case, there would be some mechanism of functional exception handling that would work with possibly-failing effects, which would assume successful execution, and handle the different types of failures separately. This would lose us guarantees about exactly which pieces of code had been executed, however. The new version of `Effects` has made handling failures much easier, and the inline guarded failure handling syntax has made code much more pleasant to write. While some extra code overhead is to be expected for verified code, I believe we can still do better.

The `PacketLang` DSL

In my mind, this is the major achievement of this project. Due to the rapid development of Idris, the code used for `PacketLang` was obsolete and not easily fixable, and as such was completely rewritten. In doing this, we have shown that the underlying theoretical concepts present in the original paper were sound in that the redesigned language (including substantial rewrites to the internals of the compiler) posed no barriers to the reimplementation of the DSL.

Additionally, newer language features such as type classes have aided in the reimplementation of the language, adding additional power.

The next logical step with regard to `PacketLang` is to provide a formal translation to the Data Definition Calculus [11], as described in Section 3.2.2.2. The DDC provides a strong underlying formalism for data description languages, and a formal translation to this calculus would likely yield dividends in terms of reasoning about its expressiveness.

`PacketLang` differs from PacketTypes [27], DataScript [1], PADS [10] and Protege [39] in that it is an *embedded* domain specific language, and does not require an external compiler or type system. In this sense, we may make use of the powerful type system provided by Idris without needing to implement something externally. These systems aim primarily to generate code for other languages to make use of otherwise unstructured data. Our approach differs in that it has more of a focus on verification: we aim to guarantee **statically** that if a packet implementation does not conform to a specification, then the program will not compile.

Through a literature survey and the results of implementing a more significant case study, we have identified two notable deficiencies in `PacketLang`. The first is that there is little scope for error reporting: a packet either parses or it fails, without any facility to report errors.

The other limitation of `PacketLang` is more technical: it is currently not possible to recursively nest `PacketLang` descriptions. To understand this, it is important to note that non-total expressions are not reduced by the type-checker, in order to preserve decidability of type-checking. Of course, in order to construct instances of, and match on, packets, we need packet specifications to reduce.

Example Applications, including DNS and Pong

While small-scale, the example applications demonstrate how the verified bindings may be used, and demonstrate that all work as intended. The end-result was more than satisfactory: using `PacketLang`, we implemented a verified version of DNS, encoding invariants within the packet description itself. We could therefore guarantee that any packet implementation accepted by the type checker was a valid DNS packet. Similarly, we could guarantee that if a packet parsed successfully, it contained valid data. We also introduced higher-level data structures to allow users to more easily construct DNS packets and work with DNS data, using dependently-typed programming patterns such as relations and GADTs to preserve the guarantees given in the specification and the higher-level data structures.

The encoding and decoding of DNS into higher-level data structures has been much simplified through the introduction of the `decodable` construct: before its introduction, a multitude of

different proofs were required in order to state seemingly obvious facts: in one instance, the code before the introduction of this construct was roughly 120 lines of inelegant boilerplate code which existed purely to appease the type checker. After the introduction of this construct, it was possible to reduce this to around 15 lines.

This emphasises the research nature of this project: at one point, a conclusion of this work would have been that `PacketLang` was not adequate for specifying more complex packet structures, whereas an additional chunk type made a huge difference to the expressiveness of the language.

Of course, limitations still exist. The biggest problem is that decoding packets remains inherently stateless, which led us to need to retain the pointer to the raw packet and re-parse parts of the packet in order to decode references. The biggest challenge comes with implementing this in a sufficiently generic way: while it would be simpler, and tempting, to reimplement `PacketLang` in a manner which would easily support DNS, this would not be general. In particular, after reimplementing and using the language, perhaps the best way of achieving this would be to shift the focus of `PacketLang` from being purely declarative, instead adding constructs for control purposes such as storing a value in a user-specified state, or providing the ability for users to specify functions to be executed by the marshalling and unmarshalling code.

The Pong case study highlights in particular the composable nature of effects: a monad transformer with this number of inner monads would be far too complex to use effectively, whereas it is achieved trivially using effects. Additionally, the Pong example serves as a demonstration that the networking and concurrency effects may be easily integrated into a larger application, and that even with simpler packets, `PacketLang` can help with transferring structured data.

Chapter 10

Conclusion

10.1 Key Achievements

This project has investigated how dependent types may be used in the domain of network programming. Moving on from the traditional usage of dependent types for the purposes of theorem proving, we have seen how Idris may be used to implement programs, and how the additional expressiveness of dependent types may be used to increase confidence in the safety of applications.

At a basic level, we have implemented a socket library that may be used by Idris developers to create networked applications. This has been accepted into the main Idris distribution, which is an open-source project.

Building on this, we have implemented TCP and UDP bindings which statically enforce resource usage protocols, preventing incorrect usage and handle leaks. In addition to allowing resource usage protocols to be enforced, these also allow for effects to more finely specified, resulting in greater safety and modularity than similar libraries in other purely functional languages.

We have also reimplemented an embedded domain-specific language to precisely specify the structure of packets, along with invariants about the data therein. By encoding invariants within the packet language description itself, we may statically guarantee that it is impossible to erroneously construct a packet which does not conform to the specification.

Through the implementation of a larger case study of DNS, we have identified ways in which dependently-typed programming patterns may be used to great effect in representing data and maintaining a congruence between lower-level precise representations and equally expressive higher-level data types. The implementation of this larger case study has also led directly to improvements to the DSL as it was originally presented.

The implementation of a networked Pong game serves to underline the composability of effects, and how they may be used to structure programs. The example shows that the different contributions of the project may all be combined to make a larger application, and that dependently-typed languages can be used to develop larger programs involving networking and concurrency.

10.2 Current Drawbacks

At the same time, the implementation of DNS has uncovered areas which require suboptimal workarounds to provide some required functionality, such as decoding references. Whilst it is still possible to decode references by retaining and passing a pointer to the packet buffer, in an ideal world this would not be necessary, and we have discussed possible ways in which this limitation may be ameliorated.

Additional concerns pertain to the amount of additional code required to handle failing cases within the effectual TCP and UDP bindings. Recent syntax improvements have ameliorated this to an extent by allowing failure cases to be handled inline instead of requiring nested case statements, but a degree of boilerplate still remains.

10.3 Future Work

Practical dependently-typed programming is an extremely young field, and there are many different avenues left to explore. Of particular interest would be the application of a DSL like `PacketLang` to specify transport-level protocols using raw sockets. This would allow the precise specification and verification of new network protocols and abstractions.

Further to specifying the *composition* of individual packets, it would be extremely interesting to see how dependently-typed techniques could be used to specify properties about the *communication* between systems, building upon the extensive existing base of literature on session types [14, 15, 13, 16]. Indeed, current work by Brady[1] implements some of this theoretical work in the context of a dependently-typed EDSL.

Session types are a very topical area of research: delving more deeply into the formal foundation behind session types, correspondences between the traditional presentation of session types as a linear language and linear logic have recently been proven [37, 23], leading to opportunities to formally prove properties such as deadlock-freedom within protocols.

Other directions may involve more elegant functional abstractions over underlying socket operations: one such abstraction would involve abstract channels being established, with the method of communication specified through the implementation of an effect handler.

10.4 Concluding Remarks

Dependent types provide a huge amount of expressive power, allowing for far more detailed reasoning to take place about programs. With recent developments, dependently-typed programming is not only becoming more powerful but more accessible: by investigating how dependent types may be used for network programming, we hope to have demonstrated how developers may benefit from these techniques to develop more precise and safe applications.

[1] `http://www.github.com/edwinb/Protocols`

Appendix A

Testing Summary

A.1 TCP

Connecting to an available remote server

Expected A connection should be established, and the client should transition into the `ClientConnected` state.

Result (Demonstrated in the echo client) – success

Attempting to connect to an unavailable remote server

Expected The connection attempt should fail, and the client should remain in the uninitialised state.

Result (Demonstrated by running the echo client without the echo server) – success

Sending and receiving string data over a connected socket

Expected The string should be sent from the client and received by the server.

Result (Demonstrated by the echo client) – success

Attempting to send string data when the server has closed the connection

Expected The send operation should fail, and the client should transition to the `ErrorState` state.

Result (Demonstrated by running the echo client and server, then terminating the server) – success

Attempting to receive string data when the client has closed the connection

Expected The receive operation should fail, and the server should transition to the `ErrorState` state.

Result (Demonstrated by running the echo client and server, then terminating the server) – success

Sending and receiving `PacketLang` data

Expected The packet should be received by the remote server.

Result (Demonstrated by the `packet` program) – success

Binding to an available address and port

Expected The operation should succeed, and the server should transition into the `ServerBound` state.

Result (Demonstrated by the echo server) – success

Attempting to bind to an unavailable addressand port

Expected The operation should fail, and the server should transition into the `ErrorState` state.

Result (Demonstrated by attempting to bind the echo server to a nonexistent address and port) – Error 13 (Permission Denied) shown; success.

A.2 UDP

Binding to an available address and port

Expected The operation should succeed, and the server should transition into the `UDPBound` state.

Result (Demonstrated by the Pong server) – success

Attempting to bind to an unavailable and port

Expected The operation should fail, and the server should transition into the `ErrorState` state.

Result (Demonstrated by attempting to bind the echo server to a nonexistent address and port) – Error 13 (Permission Denied) shown; success.

Sending string data to a nonexistent host

Expected The operation should succeed, as datagram sockets do not guarantee safe delivery.

Result (Demonstrated by the `UDPClientTest` program) – success

Sending and receiving data using datagram sockets

Expected The data should be received and displayed on the server.

Result (Demonstrated by the `UDPServerTest` program when sent a packet by the `UDPClientTest` program) – success

Sending and receiving `PacketLang` data

Expected The same packet should be received by the server that is sent by the client.

Result (Demonstrated by Pong) – success

A.3 `PacketLang`

Tests for the `PacketLang` DSL may be found in the `examples/packet/test` directory.

`IF`

Packet specification:

```
ifTest : PacketLang
ifTest = with PacketLang do
  flag <- bool
  p_if flag then cstring else bits 8
```

Packet instances:

```
ifTestInst1 : (mkTy ifTest)
ifTestInst1 = (True ## "Flag was set")

ifTestInst2 : (mkTy ifTest)
ifTestInst2 = (False ## (BInt 100 oh))
```

Output function:

```
showIfTest : (mkTy ifTest) -> String
showIfTest (True ## str) = "True, str: "++ str
showIfTest (False ## (BInt x oh)) = "False, int: "++ (show x)
```

Expected The first instance should print "True, str: Flag was set", and the second instance should print "False, 100".

Result Success

`(//)`

Packet specification:

```
eitherTest : PacketLang
eitherTest = innerTestLang1 // innerTestLang2
  where innerTestLang1 : PacketLang
        innerTestLang1 = do
          cstring
          cstring
        innerTestLang2 : PacketLang
        innerTestLang2 = bits 8
```

Packet instances:

```
eitherTestInst1 : (mkTy eitherTest)
eitherTestInst1 = (Left ("hello" ## "world"))

eitherTestInst2 : (mkTy eitherTest)
eitherTestInst2 = (Right (BInt 21 oh))
```

Output function:

```
showEitherTest : (mkTy eitherTest) -> String
showEitherTest (Left (s1 ## s2)) = "Left, s1: " ++ s1 ++ ", s2: " ++ s2
showEitherTest (Right (BInt x oh)) = "Right, int: " ++ (show x)
```

Expected The first instance should print "Left, s1: hello, s2: world", and the second instance should print "Right, int: 21".

Result Success

`LIST`

(Note: The `LIST` constructor is greedy, and as such will accept anything that it matches. For this reason, we use a slightly more complex packet specification in this test, to ensure that the data that follows is not subsumed).

Packet specification:

```
innerListStruct : PacketLang
innerListStruct = do
  b1 <- (bits 8)
  check ((val b1) < 10)

listTest : PacketLang
listTest = with PacketLang do
  list innerListStruct
  cstring
```

Packet instance:

```
listTestInst : (mkTy listTest)
listTestInst = ([((BInt 1 oh) ## oh),
  ((BInt 2 oh) ## oh), ((BInt 3 oh) ## oh)] ## "hello")
```

Output function:

```
showListTest : (mkTy listTest) -> String
showListTest (xs ## (BInt n oh)) =
  "List: " ++ (show xs) ++ ", int: " ++ (show n)
```

Expected "List: [1,2,3,], s: hello"

Result Success

LISTN

Packet specification:

```
listNTest : PacketLang
listNTest = with PacketLang do
  listn 5 cstring
  bits 8
```

Packet instance:

```
listNTestInst : (mkTy listNTest)
listNTestInst = (["This", "is", "another",
                  "PacketLang", "test"] ## (BInt 5 oh))
```

Output function:

```
showListNTest : (mkTy listNTest) -> String
showListNTest (xs ## (BInt n oh)) = "ListN: " ++
              (show xs) ++ ", int: " ++ (show n)
```

Expected "ListN: ["This", "is", "another", "PacketLang", "test"], int: 5"

Result Success

P_EQ

Packet specification:

```
eqTest : PacketLang
eqTest = with PacketLang do
  x1 <- bits 8
  x2 <- bits 8
  prop (prop_eq (val x1) (val x2))
```

Packet instance:

```
eqTestInst : (mkTy eqTest)
eqTestInst = ((BInt 5 oh) ## (BInt 5 oh) ## refl)
```

Output function:

```
showEqTest : (mkTy eqTest) -> String
showEqTest (n ## m ## p) =
  "Int1: " ++ (show $ val n) ++ ", " ++ "Int2" ++ (show $ val m) ++ ", refl"
```

Expected "Int1: 5, Int2: 5, refl'

Result Success

P_BOOL

Packet specification:

```
boolTest : PacketLang
boolTest = with PacketLang do
  b1 <- bool
  check b1
```

Packet instance:

```
boolTestInst : (mkTy boolTest)
boolTestInst = (True ## oh)
```

Output function:

```
showBoolTest : (mkTy boolTest) -> String
showBoolTest (b ## oh) = "Bool test, " ++ (show b)
```

Expected "Bool test, True"

Result Success

AND

Packet specification:

```
andTest : PacketLang
andTest = with PacketLang do
  b1 <- bool
  b2 <- bool
  prop (prop_and (prop_bool b1) (prop_bool b2))
```

Packet instance:

```
andTestInst : (mkTy andTest)
andTestInst = (True ## True ## (MkBoth oh oh))
```

Output function:

```
showAndTest : (mkTy andTest) -> String
showAndTest (True ## True ## oh) = "And test passed"
```

Expected "And test passed"

Result Success

OR

Packet specification:

```
orTest : PacketLang
orTest = with PacketLang do
  b1 <- bool
  b2 <- bool
  prop (prop_or (prop_bool b1) (prop_bool b2))
```

Packet instance:

```
orTestInst1 : (mkTy orTest)
orTestInst1 = (True ## False ## (Left oh))

orTestInst2 : (mkTy orTest)
orTestInst2 = (False ## True ## (Right oh))
```

Output function:

```
showOrTest : (mkTy orTest) -> String
showOrTest (b1 ## b2 ## (Left oh)) = "Left, b1: "
  ++ (show b1) ++ ", right: " ++ (show b2)
showOrTest (b1 ## b2 ## (Right oh)) = "Right, b1: "
  ++ (show b1) ++ ", right: " ++ (show b2)
```

Expected For the first instance, "Left, b1: True, right: False". For the second instance, "Right, b1: False, b2: True".

Result Success

A.4 DNS

Decoding the result of a DNS request, where the DNS request contains references

Expected The packet to be fully decoded, with references resolved

Result Using a packet capture known to contain backreferences:

```
0000  00 30 1b 46 bc fa 00 11  93 25 32 4b 08 00 45 00   .0.F.... .%2K..E.
0010  00 fc 9f 8b 00 00 3f 11  2a 9e 8a fb ce 02 8a fb   ......?. *.......
0020  cc ce 00 35 df fc 00 e8  9e 07 72 03 81 80 00 01   ...5.... ..r.....
0030  00 03 00 04 00 04 06 67  6f 6f 67 6c 65 02 63 6f   .......g oogle.co
0040  02 75 6b 00 00 01 00 01  c0 0c 00 01 00 01 00 00   .uk..... ........
0050  01 2c 00 04 ad c2 29 b7  c0 0c 00 01 00 01 00 00   .,....). ........
0060  01 2c 00 04 ad c2 29 b8  c0 0c 00 01 00 01 00 00   .,....). ........
0070  01 2c 00 04 ad c2 29 bf  c0 0c 00 02 00 01 00 00   .,....). ........
0080  e9 7e 00 10 03 6e 73 34  06 67 6f 6f 67 6c 65 03   .~...ns4 .google.
0090  63 6f 6d 00 c0 0c 00 02  00 01 00 00 e9 7e 00 06   com..... .....~..
00a0  03 6e 73 33 c0 5e c0 0c  00 02 00 01 00 00 e9 7e   .ns3.^.. .......~
00b0  00 06 03 6e 73 31 c0 5e  c0 0c 00 02 00 01 00 00   ...ns1.^ ........
00c0  e9 7e 00 06 03 6e 73 32  c0 5e c0 76 00 01 00 01   .~...ns2 .^.v....
00d0  00 03 8d de 00 04 d8 ef  24 0a c0 5a 00 01 00 01   ........ $..Z....
00e0  00 03 8d de 00 04 d8 ef  26 0a c0 9a 00 01 00 01   ........ &.......
00f0  00 03 8d de 00 04 d8 ef  22 0a c0 88 00 01 00 01   ........ ".......
0100  00 03 8d de 00 04 d8 ef  20 0a                     ........  .
```

Sending this to a server application (`DNSParserTest2`) decodes the result as would be expected:

```
Decoded successfully! DNS Packet:
Header: DNS Header
ID : 29187
Is query? : True
Opcode : QUERY
Is authority? : False
Is truncated? : False
Is recursion desired? : True
Is recursion available? : True
Is answer authenticated? : False
Is non-authenticated data acceptable? : False
Response : No error

QD Count: 1
AN Count: 3
NS Count: 4
AR Count: 4
Questions: [DNS Question:
DNS QNames:
["google", "co", "uk"]
DNS QType:
A
DNS QClass:
IN
]
Answers: [DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 300
DNS RR Payload: IPv4: 173.194.41.183
, DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: A
DNS RR Class: IN
```

```
DNS RR TTL: 300
DNS RR Payload: IPv4: 173.194.41.184
, DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 300
DNS RR Payload: IPv4: 173.194.41.191
]
Authorities: [DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 59774
DNS RR Payload: Domain: ["ns4", "google", "com"]
, DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 59774
DNS RR Payload: Domain: ["ns3", "google", "com"]
, DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 59774
DNS RR Payload: Domain: ["ns1", "google", "com"]
, DNS Record:
DNS RR Name: ["google", "co", "uk"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 59774
DNS RR Payload: Domain: ["ns2", "google", "com"]
]
Additionals: [DNS Record:
DNS RR Name: ["ns3", "google", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 232926
DNS RR Payload: IPv4: 216.239.36.10
, DNS Record:
DNS RR Name: ["ns4", "google", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 232926
DNS RR Payload: IPv4: 216.239.38.10
, DNS Record:
DNS RR Name: ["ns2", "google", "com"]
DNS RR Type: A
```

```
DNS RR Class: IN
DNS RR TTL: 232926
DNS RR Payload: IPv4: 216.239.34.10
, DNS Record:
DNS RR Name: ["ns1", "google", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 232926
DNS RR Payload: IPv4: 216.239.32.10
]
```

Success.

Making a request for the A record of an available domain

Expected A decoded DNS request, showing that the operation was successful and detailing the A record.

Result
(Making a request for `www.simonjf.com` using the `DNSParserTest` program) Success.

```
Header: DNS Header
ID : 1337
Is query? : True
Opcode : QUERY
Is authority? : False
Is truncated? : False
Is recursion desired? : True
Is recursion available? : True
Is answer authenticated? : False
Is non-authenticated data acceptable? : False
Response : No error

QD Count: 1
AN Count: 2
NS Count: 2
AR Count: 2
Questions: [DNS Question:
DNS QNames:
["www", "simonjf", "com"]
DNS QType:
A
DNS QClass:
IN
]
Answers: [DNS Record:
DNS RR Name: ["www", "simonjf", "com"]
```

```
DNS RR Type: CNAME
DNS RR Class: IN
DNS RR TTL: 3130
DNS RR Payload: Domain: ["simonjf", "com"]
, DNS Record:
DNS RR Name: ["simonjf", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 3130
DNS RR Payload: IPv4: 146.185.136.196
]
Authorities: [DNS Record:
DNS RR Name: ["simonjf", "com"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 157159
DNS RR Payload: Domain: ["ns38", "domaincontrol", "com"]
, DNS Record:
DNS RR Name: ["simonjf", "com"]
DNS RR Type: NA
DNS RR Class: IN
DNS RR TTL: 157159
DNS RR Payload: Domain: ["ns37", "domaincontrol", "com"]
]
Additionals: [DNS Record:
DNS RR Name: ["ns37", "domaincontrol", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 115236
DNS RR Payload: IPv4: 216.69.185.19
, DNS Record:
DNS RR Name: ["ns38", "domaincontrol", "com"]
DNS RR Type: A
DNS RR Class: IN
DNS RR TTL: 115236
DNS RR Payload: IPv4: 208.109.255.19
]
```

Making a request for the A record of a nonexistent domain

Expected A decoded DNS request, showing that there was an error and specifying a SOA record showing the authoritative name server which determined that the address did not exist.

Result

```
Decoded successfully! DNS Packet:
Header: DNS Header
```

```
ID : 1337
Is query? : True
Opcode : QUERY
Is authority? : False
Is truncated? : False
Is recursion desired? : True
Is recursion available? : True
Is answer authenticated? : False
Is non-authenticated data acceptable? : False
Response : Name error

QD Count: 1
AN Count: 0
NS Count: 1
AR Count: 0
Questions: [DNS Question:
DNS QNames:
["pines"]
DNS QType:
A
DNS QClass:
IN
]
Answers: []
Authorities: [DNS Record:
DNS RR Name: []
DNS RR Type: SOA
DNS RR Class: IN
DNS RR TTL: 5189
DNS RR Payload: SOA: DNS Start of Authority:
MName: ["a", "root-servers", "net"]
RName: ["nstld", "verisign-grs", "com"]
Serial: 2014040301
Refresh: 1800
Retry: 900
Expire: 604800
Minimum: 86400

]
Additionals: []
```

Success.

Appendix B

Status Report

The socket library provides the required functionality for using TCP and UDP. It does not yet provide more advanced functionality such as support for setting more complex socket options. Additionally, IPv6 is not yet supported.

The TCP and UDP bindings work as intended, and can send both string data and data encoded using `PacketLang` over the network. Future work here would involve making more elegant abstractions, for example using streams for TCP sockets.

The `PacketLang` DSL works correctly, and allows for specifications to be written, and for data to be read and written according to these specifications. Future work would investigate whether it is possible to do recursive definitions, and implement better error reporting and other constructs.

DNS supports a reasonable subset of the available types, however future work would extend this further. the data received, and the length of the data.

Appendix C

User Manual

C.1 Installation and Usage

To install the library, you will most likely need the latest Git HEAD version of Idris. You can get this by cloning http://www.github.com/idris-lang/Idris-dev.

Once Idris is installed, the library may be installed by running `idris -install idrisnet.ipkg`.

To use the library, it is also necessary to include the `-p effects` and `-p network` flags when compiling.

C.2 Socket Library

Listing C.1: `IdrisNet.Socket API`

```
socket : SocketFamily ->
         SocketType ->
         ProtocolNumber ->
         IO (Either SocketError Socket)

close : Socket -> IO ()
bind : Socket -> (Maybe SocketAddress) -> Port -> IO Int
connect : Socket -> SocketAddress -> Port -> IO Int
listen : Socket -> IO Int
send : Socket -> String -> IO (Either SocketError ByteLength)
recv : Socket -> Int -> IO (Either SocketError (String, ByteLength))
sendBuf : Socket -> Ptr -> Int -> IO (Either SocketError ByteLength)
recvBuf : Socket -> Ptr -> Int -> IO (Either SocketError ByteLength)
accept : Socket -> IO (Either SocketError (Socket, SocketAddress))
sendTo : Socket -> SocketAddress -> Port ->
  String -> IO (Either SocketError ByteLength)
sendToBuf : Socket -> SocketAddress -> Port ->
  BufPtr -> ByteLength -> IO (Either SocketError ByteLength)
recvFrom : Socket -> ByteLength ->
  IO (Either SocketError (UDPAddrInfo, String, ByteLength))
recvFromBuf : Socket -> BufPtr -> ByteLength ->
  IO (Either SocketError (UDPAddrInfo, ByteLength))
accept : Socket -> IO (Either SocketError (Socket, SocketAddress))
```

Listing C.1 shows the `IdrisNet.Socket` API.

C.2.1 Socket Creation

To create a socket, use the `socket` function, specifying a `SocketFamily`, `SocketType` and protocol number.

Values of type `SocketFamily` may be constructed using the following constructors:

```
data SocketType = NotASocket  -- Not a socket, used in certain operations
                | Stream      -- TCP
                | Datagram    -- UDP
                | RawSocket   -- Raw sockets.
```

Values of type `SocketType` may be constructed using the following constructors:

```
data SocketFamily = AF_UNSPEC -- Unspecified
                  | AF_INET   -- IP / UDP etc. IPv4
                  | AF_INET6  -- IP / UDP etc. IPv6
```

C.2.2 Closing a Socket

Simply call `close` with the socket you wish to close.

C.2.3 Binding

To bind, call the `bind` function. If you wish to specify the address to which to bind, specify this wrapped in the `Just` constructor. If `Nothing` is specified, then the library will automatically bind on an available local address.

C.2.4 Listening

Call `listen` with the socket on which you wish to listen.

C.2.5 Accepting Clients

To accept a client, call `accept` on a listening socket. This will either return the newly-created socket and an associated socket address, or an error.

C.2.6 Sending Data

To send string data on a connected stream socket, call `send` with the socket and string as arguments. To send string data on a datagram socket, call `sendTo` with the socket address and port as arguments.

Socket addresses are specified using the `SocketAddress` data type:

```
data SocketAddress = IPv4Addr Int Int Int Int
                   | Hostname String
```

To send data from a buffer, use the `sendBuf` and `sendToBuf` functions.

C.2.7 Receiving Data

To receive data into a buffer, use the `recvBuf` and `recvFromBuf` functions.

C.3 TCP Client

C.3.1 Connecting

To connect to a remote host, use the `tcpConnect` function, specifying the `SocketAddress` and port to use.

It is then necessary to check for failure as so:

```
OperationSuccess _ <- tcpConnect sa p
  | RecoverableError err => ... -- A recoverable error (() state)
  | FatalError err => ... -- A fatal error (() state)
  | ConnectionClosed => ... -- (() state)
... -- The rest of the program, in the \texttt{ClientConnected} state.
```

C.3.2 Sending and Receiving Data

To send string data, use the `tcpSend` function, specifying the string to send. To send a `PacketLang` packet, use the `tcpWritePacket` function, specifying the `PacketLang` specification and a concrete instance of the packet.

To receive string data, use the `tcpRecv` function, specifying the buffer size. To receive a `PacketLang` packet, use the `tcpReadPacket` function, specifying the `PacketLang` description and the buffer size.

You must check for failure after each operation, as described above.

C.4 TCP Server

C.4.1 Binding

To bind to a socket address and port, use the `bind` function, specifying an address if needed. If not, specify `Nothing` and the address will be chosen automatically.

C.4.2 Accepting Clients

To accept a new client, you will need to create a function of type `ClientProgram`, which will run using the newly-acquired socket. Pass this as an argument to the `accept` function. It is then possible to communicate with this socket using the functions described above.

```
ClientProgram : Type -> Type
ClientProgram t = {[TCPSERVERCLIENT (ClientConnected)] ==>
                   [TCPSERVERCLIENT ()]} Eff IO t
```

C.4.3 Closing the Server Socket

To close a bound socket, use the `closeBound` function. To close a listening socket, use the `closeListening` function. To close a socket in an error state, use the `finaliseServer` function.

C.5 UDP Client

To send data using the UDP client effect, simply use the `udpWriteString` and `udpWritePacket` functions.

The `udpWriteString` function takes the remote address, port, and string to send as its arguments. The `udpWritePacket` function takes the remote address, port, a `PacketLang` description and an instance of that packet.

C.6 UDP Server

In order to send and receive data using a UDP server socket, it is first necessary to bind to a local address and port. This is done using the `udpBind` function, which takes a `Maybe SocketAddress` and port number as its arguments.

It is then necessary to check that the operation succeeded, as with TCP.

Once the socket has been successfully bound, it is then possible to use the `udpWriteString` and `udpWritePacket` functions to write strings and `PacketLang` packets respectively, and `udpReadString` and `udpReadPacket` to read data. These functions return, if successful, a tuple of the address information of the remote client, the data received, and the length of the data.

C.7 PacketLang

To specify a `PacketLang` specification, make a function of type `PacketLang`. You may then construct a specification according to the grammar in Section 7.3.2.

An example would be:

```
helloWorld : PacketLang
helloWorld = do
  cstring
  cstring
```

To specify an implementation, make a function of type `mkTy pl`, where `pl` is the name of the `PacketLang` specification. To continue the `helloWorld` example:

```
helloWorldInst : (mkTy helloWorld)
helloWorldInst = ("hello" ## "world")
```

Separate data items using the `##` operator.

Bibliography

[1] Godmar Back. DataScript - A Specification and Scripting Language for Binary Data. In *In Generative Programming and Component Engineering*, pages 66–77, 2002. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.2378.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[3] Edwin Brady. Interactive Idris Editing with Vim. http://edwinb.wordpress.com/2013/10/28/interactive-idris-editing-with-vim/.

[4] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. *SIGPLAN Not.*, 48(9):133–144, September 2013. ISSN 0362-1340. doi: 10.1145/2500365.2500581. URL http://dx.doi.org/10.1145/2500365.2500581.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013. ISSN 1469-7653. doi: 10.1017/s095679681300018x. URL http://dx.doi.org/10.1017/s095679681300018x.

[6] Edwin C. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[7] Edwin C. Brady. IDRIS —: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0. doi: 10.1145/1929529.1929536. URL http://dx.doi.org/10.1145/1929529.1929536.

[8] Peter Dybjer and Anton Setzer. A Finite Axiomatization of Inductive-Recursive Definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer Berlin Heidelberg, 1999. doi: 10.1007/3-540-48959-2_11. URL http://dx.doi.org/10.1007/3-540-48959-2_11.

[9] D. Eastlake. RFC 2535–Domain Name System Security Extensions. 1999.

[10] Kathleen Fisher and Robert Gruber. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 40 of *PLDI '05*, pages 295–304, New York, NY, USA, June 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065046. URL http://dx.doi.org/10.1145/1065010.1065046.

[11] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111039. URL http://dx.doi.org/10.1145/1111037.1111039.

[12] Simon Fowler and Edwin Brady. Dependent Types for Safe and Secure Web Programming. In Rinus Plasmeijer, editor, *Implementation and Application of Functional Languages*, 2013.

[13] Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer Berlin Heidelberg, 1999. doi: 10.1007/3-540-49099-x_6. URL http://dx.doi.org/10.1007/3-540-49099-x_6.

[14] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993. doi: 10.1007/3-540-57208-2_35. URL http://dx.doi.org/10.1007/3-540-57208-2_35.

[15] Kohei Honda, VascoT Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, chapter 9, pages 122–138. Springer Berlin Heidelberg, Berlin/Heidelberg, 1998. ISBN 3-540-64302-8. doi: 10.1007/bfb0053567. URL http://dx.doi.org/10.1007/bfb0053567.

[16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328472. URL http://dx.doi.org/10.1145/1328897.1328472.

[17] Simon P. Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple Unification-based Type Inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, September 2006. ISSN 0362-1340. doi: 10.1145/1160074.1159811. URL http://dx.doi.org/10.1145/1160074.1159811.

[18] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in Action. *SIGPLAN Not.*, 48 (9):145–158, September 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500590. URL http://dx.doi.org/10.1145/2544174.2500590.

[19] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[20] Xavier Leroy. The OCaml programming language. 1998.

[21] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[22] Sam Lindley and Conor McBride. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 81–92, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. doi: 10.1145/2503778.2503786. URL http://dx.doi.org/10.1145/2503778.2503786.

[23] Sam Lindley and J. Garrett Morris. Sessions as propositions. 2014.

[24] James Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.

[25] Conor McBride. Faking it Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, July 2002. ISSN 1469-7653. doi: 10.1017/s0956796802004355. URL http://dx.doi.org/10.1017/s0956796802004355.

[26] Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, May 2007. doi: 10.1017/s0956796807006326. URL http://dx.doi.org/10.1017/s0956796807006326.

[27] Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM. ISBN 1-58113-223-9. doi: 10.1145/347059.347563. URL http://dx.doi.org/10.1145/347059.347563.

[28] Paul Mockapetris. RFC 1035–Domain names–implementation and specification. 1987.

[29] Ulf Norell. Towards a practical programming language based on dependent type theory. 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.7934.

[30] Nicolas Oury and Wouter Swierstra. The Power of Pi. *SIGPLAN Not.*, 43(9):39–50, September 2008. ISSN 0362-1340. doi: 10.1145/1411203.1411213. URL http://dx.doi.org/10.1145/1411203.1411213.

[31] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_7. URL http://dx.doi.org/10.1007/978-3-642-00590-9_7.

[32] S. L. P. Simon and Philip Wadler. Imperative Functional Programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.2504.

[33] The Wikimedia Foundation. Iterative Development Model. http://en.wikipedia.org/wiki/File:Iterative_development_model_V2.jpg, .

[34] The Wikimedia Foundation. Rapid Application Development. http://en.wikipedia.org/wiki/File:RADModel.JPG, .

[35] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997. ISSN 01636804. doi: 10.1109/35.565655. URL http://dx.doi.org/10.1109/35.565655.

[36] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143169. URL http://dx.doi.org/10.1145/143165.143169.

[37] Philip Wadler. Propositions As Sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364568. URL http://dx.doi.org/10.1145/2398856.2364568.

[38] Yan Wang and Verónica Gaspes. A Library for Processing Ad hoc Data in Haskell. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-24452-0_10. URL http://dx.doi.org/10.1007/978-3-642-24452-0_10.

[39] Yan Wang and Verónica Gaspes. An embedded language for programming protocol stacks in embedded systems. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 63–72, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0485-6. doi: 10.1145/1929501.1929511. URL http://dx.doi.org/10.1145/1929501.1929511.

[40] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon P. Jones, Dimitrios Vytiniotis, and José Pedro Magalh aes. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. URL http://dx.doi.org/10.1145/2103786.2103795.