IDRIS — Systems Programming meets Full Dependent Types

Edwin Brady
School of Computer Science, University of St Andrews, St Andrews, Scotland.
Email: eb@cs.st-andrews.ac.uk

Abstract
Dependent types have emerged in recent years as a promising approach to ensuring program correctness. However, existing dependently typed languages such as Agda and Coq work at a very high level of abstraction, making it difficult to map verified programs to suitably efficient executable code. This is particularly problematic for programs which work with bit level data, e.g. network packet processing, binary file formats or operating system services. Such programs, being fundamental to the operation of computers in general, may stand to benefit significantly from program verification techniques. This paper describes the use of a dependently typed programming language, IDRIS, for specifying and verifying properties of low-level systems programs, taking network packet processing as an extended example. We give an overview of the distinctive features of IDRIS which allow it to interact with external systems code, with precise types. Furthermore, we show how to integrate tactic scripts and plugin decision procedures to reduce the burden of proof on application developers. The ideas we present are readily adaptable to languages with related type systems.

1. Introduction
Systems software, such as an operating system or a network stack, underlies everything we do on a computer, whether that computer is a desktop machine, a server, a mobile phone, or any embedded device. It is therefore vital that such software operates correctly in all situations. Dependent types have emerged in recent years as a promising approach to ensuring the correctness of software, with high level verification tools such as Coq [8] and Agda [24] being used to model and verify a variety of programs including domain-specific languages (DSLs) [25], parsers [9], compilers [16] and algorithms [32]. However, since these tools operate at a high level of abstraction, it can be difficult to map verified programs to efficient low level code. For example, Oury and Swierstra’s data description language [25] works with a list of bits to describe file formats precisely, but it does not attempt to store concrete data compactly or efficiently.

This paper explores dependent type based program verification techniques for systems programming, using the IDRIS programming language. We give an overview of dependent type based program verification techniques for systems programming, using the IDRIS programming language. We give an overview of the key features of which distinguish it from other related languages and give an extended example of the kind of program which stands to benefit from type-based program verification techniques — a data description language influenced by PADS [19] and PACKETTYPES [22]. This language is an embedded domain-specific language (EDSL) [14] — that is, it is implemented by embedding in a host language, exploiting the host’s parser, type system and code generator. In this EDSL, we can describe data formats at the bit level, as well as express constraints on the data. We implement operations for converting data between high level IDRIS data types and bit level data, using a foreign function interface which gives IDRIS types to C functions. This language has a serious motivation: we would like to implement verified, efficient network protocols [1]. Therefore we show two packet formats as examples: Internet Control Message Protocol (ICMP) packets, and Internet Protocol (IP) headers.

1.1 Contributions
The main contribution of this paper is to demonstrate that a high level dependently typed language is capable of implementing and verifying code at a low level. We achieve this in the following specific ways:

• We describe the distinctive features of IDRIS which allow integration of low level systems programming constructs with higher level programs verified by type checking (Section 2).
• We show how an effective Foreign Function Interface can be embedded in a dependently typed language (Section 2.6).
• We introduce a serious systems application where a programming language meets program verification, and implement it fully: a binary data description language, which we use to describe ICMP and IP headers precisely, expressing the data layout and constraints on that data (Section 3).

We show how to tackle some of the awkward problems which can arise in practice when implementing a dependently typed application. These problems include:

• Dealing with foreign functions which may have more specific inputs and outputs than their C types might suggest — e.g. we might know that an integer may lie within a specific range.
• Satisfying proof obligations which arise due to giving data and functions precise types. As far as possible, we would like proof obligations to be solved automatically, and proof requirements should not interfere with a program’s readability.

2. An Overview of IDRIS
IDRIS is an experimental functional programming language with dependent types, similar to Agda [24] or Epigram [7, 21]. It is eagerly evaluated and compiles to C via the Epic supercombinator compiler1. IDRIS has monadic I/O in the style of Hancock and Setzer [13], and a simple foreign function interface. It is implemented

[Copyright notice will appear here]
on top of the IVOR theorem proving library [4], giving direct access to an interactive tactic-based theorem prover. In this section, we give an overview of the main features of DRIS. It is not intended as a tutorial on dependently typed programming, or even DRIS specifically, but rather to show in particular the features which allow program verification to meet practical systems programming.

### 2.1 Simple Types and Functions

DRIS data types are declared using a similar syntax to Haskell data types. However, DRIS syntax is not whitespace sensitive and declarations must end with a semicolon. For example, natural numbers, an option type and lists are declared as follows in the standard library:

```haskell
data Nat = O | S Nat;
data Maybe a = Nothing | Just a;
data List a = Nil | Cons a (List a);
```

Functions are implemented by pattern matching, again using a single colon (rather than Haskell’s double colon `::`). For example, addition on natural numbers can be defined as follows, again taken from the standard library:

```haskell
plus : Nat -> Nat -> Nat;
plus O y = y;
plus (S k) y = S (plus k y);
```

Additionally, DRIS has a number of **primitive** types, summarised in Table 1. Note in particular that `String` is a primitive, rather than a sequence of characters, for efficiency reasons. `Ptr` is used as a simple means of referencing C values.

### 2.2 Dependent Types

Dependent types allow types to be predicated on values. DRIS uses *full-spectrum* dependent types, meaning that there is no restriction on which values may appear in types. For example, `Vectors` are lists which carry their size in the type. They are declared as follows in the standard library, using a syntax similar to that for Generalised Algebraic Data Types (GADTs) in Haskell [27]:

```haskell
infixr $ ::;

data Vect : Set -> Nat -> Set where
  VNil : Vect a O
  | (::) : a -> Vect a k -> Vect a (S k);
```

This declares a family of types. We explicitly state the type of the type constructor `Vect` — it takes a type and a Nat as an argument, and returns a new type. `Set` is the type of types. We say that `Vect` is parameterised by a type, and indexed over Nats. Each constructor targets a different part of the family — `VNil` can only be used to construct vectors with zero length, and `::` to construct vectors with non-zero length.

Note also that we have defined an infix operator, `::`, and declared it as right associative with precedence 5. Functions, data constructors and type constructors may all be given infix operators as names.

We can define functions on dependent types such as `Vect` in the same way as on simple types such as `List` and `Nat` above, by pattern matching. The type of a function over `Vect` will describe what happens to the lengths of the vectors involved. For example, `vappend` appends two `Vect`s, returning a vector which is the sum of the lengths of the inputs:

```haskell
vappend : Vect a n -> Vect a m -> Vect a (plus n m);
vappend VNil VNil = VNil;
vappend (x :: xs) ys = x :: vappend xs ys;
```

**Implicit Arguments**

In the definition of `vappend` above, we have used undeclared names in the type (e.g. `a` and `n` in `Vect a n`). The type checker still needs to infer types for these names, and add them as *implicit* arguments. The type of `vappend` could alternatively be written as:

```haskell
vappend : (a:Set) -> {n:Nat} -> {m:Nat} -> Vect a n -> Vect a m -> Vect a (plus n m);
```

The braces `{}` indicate that the arguments `a`, `n` and `m` can be omitted when applying `vappend`. In general, if an undeclared name appears in an index to a type (i.e. `a`, `n` and `m` in `vappend`) DRIS will automatically add it as an implicit argument, and attempt to infer its type.

**using notation**

If the ordering of implicit arguments is important (e.g. if one depends on another) it is necessary to declare the implicit arguments manually. The `Elem` predicate, for example, is a type which states that `x` is an element of a vector `xs`:

```haskell
data Elem : a -> Vect a n -> Set where
  here : {x:a} -> {xs:Vect a n} ->
    Elem x (x :: xs)
  there : {x,y:a} -> {xs:Vect a n} ->
    Elem x xs -> Elem x (y :: xs);
```

We give the implicit arguments `x` and `y` because they depend on another implicit argument `a`, and `xs` because it depends on two implicit arguments `a` and `n`. DRIS will add `a` and `n` automatically before the arguments which were given manually.

To avoid excessive repetition (as with `x`, `y` and `xs` above) and to improve the clarity when reading type declarations, DRIS provides the using notation:

```haskell
using (x:a, y:a, xs:Vect a n) {
  data Elem : a -> Vect a n -> Set where
    here : Elem x (x :: xs)
    | there : Elem x xs -> Elem x (y :: xs);
}
```

The notation `using (x:a) {...}` means that in the block of code, if `x` appears in a type it will be added as an implicit argument with type `a`.

**Termination Checking**

In order to ensure termination of type checking (and therefore its decidability), we must distinguish terms for which evaluation definitely terminates, and those for which it may not. We take a simple but effective approach to termination checking: any functions that do not satisfy a simple syntactic constraint on recursive calls will not be reduced by the type checker. The constraint we use is that each recursive call must have an argument that is structurally smaller than the input argument in the same position, and that these arguments must belong to a strictly positive data type. We check for

---

2 A tutorial is available at http://www.idris-lang.org/tutorial/
This defines a macro \( f \) simpler syntax, and take the following form:

```
Syntax macros allow more complex syntactic forms to be given.
```

We can use a Haskell-like list notation:

```
(a & b)
```

One common data type is the `Pair`:

```
data Pair a b = mkPair a b;
```

Pair types can instead be abbreviated to \((a & b)\), with pair values written \((a, b)\). 

```
Dependent pairs are declared as follows:
```

```
data Sigma : (A:Set)->(P:A->Set)->Set where
  Exists : {P:A->Set} -> (a:A) -> P a -> Sigma A P;
```

Dependent pair types can be abbreviated to \((x : a \;**\; P \; x)\), which stands for \(\Sigma a \; (\forall x . P \; x)\). A value in a dependent pair is written \(\langle a, b \rangle\). Often, the first value can be inferred from the type of the second, so the value is written \(\langle \ldots, b \rangle\).

```
IDRS also provides a more concise notation for Lists. Instead of constructing a list using nested applications of Cons...
```

```
Cons a (Cons b (Cons c ... ))
```

...we can use a Haskell-like list notation:

```
[a, b, c, ...]
```

### Syntax Macros

Syntax macros allow more complex syntactic forms to be given simpler syntax, and take the following form:

```
syntax f x1 x2 ... xn = e
```

This defines a macro \( f \) with \( n \) arguments \( x1 \) to \( xn \). Wherever an \( f \) appears in a program (whether a pattern or an expression), with \( n \) arguments, it will be replaced with \( e \), substituting \( x1 \) to \( xn \) in \( e \). Macros are type checked at the point they are expanded. It is an error to apply the macro with fewer than \( n \) arguments.

There are two main uses of syntax macros. Firstly, since they are applied in programs as well as programs, they can be used as pattern synonyms. Secondly, when implementing embedded languages they can be used to define a clearer syntax, taking advantage of the fact that they are not type checked until they are expanded. In particular, they are useful for hiding proof obligations as we will see in Section 2.8.

### The with rule

Very often, we need to match on the result of an intermediate computation. IDRS provides a construct for this, the `with` rule, modelled on a similar construct in Epigram [21] and Agda [24]. It takes account of the fact that matching on a value in a dependently typed language can affect what we know about the forms of other values. In its simplest form, the `with` rule adds another argument to the function being defined, e.g. the vector filter function, defined as follows:

```
vfilter : (a -> Bool) -> Vect a n -> (p ** Vect a p);
vfilter p VNil = <| _ , VNil |>
vfilter p (x :: xs) with vfilter p xs {  
  | <| _ , xxs' |> = if (p x) then <| _ , x :: xxs' |>  
  else <| _ , xxs' |>;
}
```

If the intermediate computation itself has a dependent type, then the result can affect the forms of other arguments — we can learn the form of one value by testing another. For example, a `Nat` is either even or odd. If it is even it will be the sum of two equal `Nats`. Otherwise, it is the sum of two equal `Nats` plus one:

```
data Parity : Nat -> Set where
  even : Parity (plus n n)  
  odd : Parity (S (plus n n));
```

We say `Parity` is a view of `Nat`. It has a covering function which tests whether its input is even or odd and constructs the predicate accordingly:

```
parity : (n:Nat) -> Parity n;
```

We can use this to write a function which converts a natural number to a list of binary digits (least significant first) as follows, using the `with` rule:

```
natToBin : Nat -> List Bool; 
natToBin 0 = Nil; 
natToBin k with parity k {  
natToBin (plus j j) | even = Cons False (natToBin j);  
natToBin (S (plus j j)) | odd = Cons True (natToBin j);
}
```

The value of the result of `parity k` affects the form of `k`, because the result of `parity k` depends on `k`. Therefore as well as the patterns for the result of the intermediate computation (even and odd) we also state how the results affect the other patterns. Note that there is a function in the resulting patterns (`plus`) and repeated occurrences of `j` — this is permitted since the form of these patterns is determined by the form of `parity k`.

### Evaluation and Compilation

IDRS consists of an interactive environment with a read-eval-print loop (REPL) and a compiler. The interactive environment allows inspection of types and values and provides an interface to the theorem proving tools in IVOR[4]. For example, at the IDRS prompt we can evaluate:

```
$ idris test.idr
Idris> natToBin (intToNat 6) 
[False, True, True] : List Bool
```

With the `:t` command, we can check types:

```
Idris> :t Elem O 
Vec Nat y0 -> Set
```

With the `:d` command (standing for “definition”), we can inspect the internal form of a function or data structure. This is useful to see how optimisations such as forcing and collapsing [3, 6] and partial evaluation [5] affect the compiled representation. Vectors, for example, need not store their length:

```
Idris> :d Vect 
Vec constructors:
  VNil 
  (::) a (Vec a k)
```

A proof that an item is an element of a list reduces to a natural number (which has an optimised representation) since it corresponds to the index at which the item appears:

```
Idris> :d Elem 
Elem constructors: 
  0 
  S (Elem x xs)
```

Some types, such as the following “less than” predicate `LT` on `Nats`, carry no run-time information at all, so have no run-time representation:
I/O does not do so directly, but rather generates an I/O tree by a continuation that defines how to process the response to that command: zero, where an I/O operation consists of a command followed by a continuation that determines how to process the response to that command.

2.6 I/O system and Foreign Functions

Input and output in I/DRIS, like Haskell, is achieved using an I/O monad. This is implemented in the style of Hancock and Setzer [13], where an I/O operation consists of a command followed by a continuation that determines how to process the response to that command:

```
data IO : Set -> Set where
  IODo : (Command -> IO (IO a)) -> IO a;
  IOReturn : a -> (IO a)
```

In this way, we preserve purity in that a program which performs I/O does not do so directly, but rather generates an I/O tree which describes the actions which will be executed when the program is run. Commands include a number of primitive operations, for example:

```
data Command : Set where
  PutStr : String -> Command
  GetStr : Command
  ... ;
```

A Response to a command is provided by the run-time system. Executing each Command gives a response of an appropriate type:

```
Response : Command -> Set;
Response (PutStr s) = ();
Response GetStr = String;
...
```

We define the usual bind and return operations:

```
(>>=) : IO a -> (a -> IO b) -> IO b;
(>>=) (IODo c p) k = IODo c (
y => (bind (p y) k));
return : a -> IO a;
return x = IOReturn x;
```

do-notation

Rather than using the (>>=) operator to sequence I/O operations, I/DRIS provides do-notation, like Haskell, which expands to the (>>=) and return functions by default. For example:

```
greet : IO ()
greet = do { putStrLn "What is your name? ";
name <- getStr;
putStrLn ("Hello " ++ name) ; }
```

Unlike Haskell, however, we do not (yet) provide a Monad type class for more general do-notation. Instead, we allow do-notation to be rebound locally. For example, we can write a bind operation for Maybe as follows:

```
maybeBind : Maybe a -> (a -> Maybe b) -> Maybe b;
maybeBind Nothing f = Nothing;
maybeBind (Just x) f = f x;
```

For the return operation, we can use Just. We can use these inside a do block with a do using declaration, which takes the bind and return operations as parameters. For example, a function which adds two Maybe Ints, using do-notation, could be written as follows:

```
do using (maybeBind, Just) {
  m_add : Maybe Int -> Maybe Int -> Maybe Int;
m_add x y = do { x' <- x;
y' <- y;
  return (x' + y'); }
}
```

This is, however, a temporary solution. We plan to implement type classes for overloading functions following the style of Sozeau and Oury in Coq [31].

Foreign Functions

When an I/O program is compiled, the compiler generates code for each of the Commands using appropriate standard C library functions. Conveniently, using this approach to I/O we can describe foreign functions directly in I/DRIS without introducing any language extensions, deferring details of how the functions are executed (and how values are marshaled to and from C) to the run-time system.

We begin by defining a universe of types FType which can be passed to foreign functions, and a decoding function i_ftype:

```
data FType = FUnit | FInt | FStr | FPtr | FAny Set;
```

Each of these types has a representation in C; these are int, char*, and void* (or in practice, any pointer type) for FInt, FStr and FPtr respectively. For FAny, we use the run-time systems internal representation of values. FUnit is used as the return type of a void function. Then a foreign function has an external name, a list of argument types, and a return type:

```
data ForeignFun = FFun String (List FType) FType;
```

To call a foreign function, we will need a concrete list of arguments corresponding to the expected argument types. We will also find it convenient to be able to append foreign argument lists:

```
using (xs,ys:List FType) {
data FArgList : List FType -> Set where
  fNil : FArgList Nil
  fCons : i_ftype x -> FArgList xs ->
  FArgList (Cons x xs);

  fapp : FArgList xs -> FArgList ys ->
  FArgList (xs ++ ys);
  fapp (fCons fx xs) ys = fCons fx (fapp fx ys);

fopen : ForeignFun;
open_fun : ForeignFun "fopen" [FStr, FStr] FPtr;
```

In order to run such functions, we need to extend the Command type, and correspondingly the Response function, with foreign function descriptions. A foreign function call takes the function description and a concrete argument list, and the response gives us a value of the declared return type.
Using foreign functions involves creating a description with mkForeign. The associated with calling foreign functions is the marshaling between C to direct execution of the I/O operations — the only overhead associated with an I/O tree. In practice, for efficiency, an I/O tree is compiled into compiled code that executes it. Conceptually, the description of the external function is held in the Foreign constructor.

```haskell
mkForeign (FFun fn args rt) = mkFType f;
```

For example, mkFType fopen gives `String -> String -> IO File;`

```haskell
fopen : String -> String -> IO File;
fopen str mode = do { ho <- _fopen str mode;
  return (FHandle ho); }
```

Similarly, mkFType fclose gives `File -> IO ();`

```haskell
fclose : File -> IO ();
fclose (FHandle h) = _fclose h;
```

Finally, we'd like to give foreign functions a higher level I/O tree.

```haskell
fread : File -> IO String;
fread (FHandle hn) = _fread hn;
```

2.7 Metavariables and theorem proving

Sooner or later when programming with dependent types, the need to prove a theorem arises. This typically happens when using an indexed data type, if a value's index does not match the required type but is nevertheless provably equal. Since Idris is built on a tactic based theorem proving library, IVOR, we are able to provide access to the tactic engine. Suppose we would like to prove the following theorem about plus:

```haskell
plusReducesO : (n:Nat) -> n = plus n 0;
```

We can declare just the type, and prove the theorem interactively in the Idris environment. The :p command enters the interactive proof mode and displays the current Goal:

```
Idris> :p plusReducesO
```

```
plusReducesO : (n:Nat) -> n = plus n 0;
```

In proof mode, we are given a list of premisses (initially empty) above the line, and the current goal (named H0 here) below the line. At the prompt, we can enter tactics to direct the construction of a proof. Here, we use the intro tactic to introduce the argument n as a premiss, followed by induction on n.

```
Idris> intro
```

```
n : Nat
```

```
H0 ? (n : Nat) -> n = plus n 0
```

The resulting goal `0 = plus 0 0` can be solved by reflexivity (the refl tactic), since plus 0 0 normalises to 0. This leaves the inductive case:

```
Idris> refl
```

```
n : Nat
```

```
H1 ? (k:Nat) -> (k = plus k 0) -> (S k = plus (S k) 0)
```

```
plusReducesO> intro k, ih
```

```
n : Nat
k : Nat
ih : k = plus k 0
```

```
H1 ? (k : Nat) -> (k = plus k 0) -> (S k = plus (S k) 0)
```

If we reduce this goal to normal form (using the compute tactic) we can apply the inductive hypothesis ih, and complete the proof by reflexivity.
Finally, entering `qed` will verify the proof and output a log of the proof script which can be pasted directly into the source file.

```
plusReducesD> rewrite ih
```

\(n : \text{Nat}
\)
\(k : \text{Nat}
\)
\(ih : k = \text{plus} k 0
\)

```
-----------------------------------------------------------------------
H3 \(S k = S k
\)
```

\(\text{plusReducesD> refl}
\)

No more goals

Finally, entering `qed` will verify the proof and output a log of the proof script which can be pasted directly into the source file.

```
plusReducesD> proof {
%intro; %induction n; %intro k,ih;
%compute; %rewrite ih; %refl; %qed;
};
```

Proof by Pattern Matching

We can also write proofs by writing pattern matching functions. Often this is more convenient and easier to read than a purely tactic based proof. Suppose for example we wish to prove that appending two lists is associative:

```
app_assoc :
  (xs:List a) -> (ys:List a) -> (zs:List a) ->
  (xs ++ (ys ++ zs) = (xs ++ ys) ++ zs);
```

\((++)\) is implemented in the library by pattern matching and recursion on its first argument. A good approach to proving a function's properties is often to follow the structure of the function itself. It is therefore easier to break this down by writing down a pattern matching definition, and using the theorem prover to add the final details. We can do this by leaving holes (or metavariables) in the proof, marked by `?name`:

```
app_assocNil ys zs = ?app_assocNil;
app_assoc (Cons x xs) ys zs = let ih = app_assoc xs ys zs in ?app_assocCons;
```

In the `Cons` case we have gained access to an induction hypothesis by making a recursive call corresponding to the recursive call in the definition of \((++)\). When we invoke `IDRIS`, it will report which proof obligations are to be satisfied:

```
Proof obligations:
[app_assocNil,app_assocCons]
```

These can be proved using the tactic engine, and the scripts pasted in as before. In practice, it is often clearer to write definitions with metavariables as above, which give a "sketch" of a proof, and leave the remaining proof scripts with the precise details to the end of a program file. In some ways, this resembles a paper which gives the outline of a proof and postpones the details until an appendix or an accompanying technical report.

2.8 Plugin Decision Procedures

Many proof obligations which arise in practice are solved by straightforward applications of simple rewriting rules (e.g. applications of list associativity) or by applying known decision procedures such as a Presburger arithmetic solver [30]. In these cases, we would like to avoid writing proofs by hand and where appropriate we might prefer to hide from the programmer the fact that a proof was required at all. IDRIS provides two basic mechanisms for achieving this: firstly, a `decide` tactic, which applies a decision procedure implemented in IDRIS; and secondly, an embedded domain specific language for constructing user defined tactics.

decide tactic

The `decide` tactic, when given a goal of the form \(T a b c\), and a function \(f : (a:A) -> (b:B) -> (c:C) -> \text{Maybe} (T a b c)\), will apply \(f\) to \(a\ b\ c\). If the result is \text{Just} \(p\), it will solve the goal with \(p\). For example, we could write a function which determines whether a value is an element of a list and returns a proof of the `Elem` predicate if so:

```
isElem : (x:a) -> (xs:Vect a n) -> \text{Maybe} (Elem x xs);
```

If the relevant values are statically known — which is not unlikely when implementing an EDSL for example [5] — the `decide` tactic could fill in required proofs of `Elem` automatically using `isElem`.

Tactic EDSL

```
data Tactic : Set where
  Fill : {a:Set} -> a -> Tactic
  Refine : String -> Tactic
  Trivial : Tactic
  Decide : {a:Set} -> Maybe a -> Tactic
  SearchContext : Tactic
  Try : Tactic -> Tactic -> Tactic
  Fail : String -> Tactic
  ... ;
```

Figure 2. EDSL for Tactic Construction

More generally, when the relevant values are not statically known, a simple decision procedure will not be suitable. In such cases, we can dynamically construct a tactic script using an EDSL for tactic construction similar to Coq's `Ltac` language [8], a fragment of which is shown in Figure 2. Each constructor corresponds to a tactic, or a means of combining tactics. Try, for example, applies the first tactic, and if that fails, applies the second.

The `decide` tactic, when given a goal of the form \(T a b c\), and a function \(f : (a:A) -> (b:B) -> (c:C) -> \text{Tactic}\), will apply \(f\) to \(a\ b\ c\) and execute the resulting tactic. For example, we can write a slightly extended version of a tactic to search for proofs of vector membership as follows:

```
isElemTac : a -> Vect a n -> Tactic;
isElemTac x xs = Try (Decide (isElem x xs))
  (Try SearchContext (Fail "Can't find element");
```

This tactic first tries to apply the `isElem` proof search. If it succeeds, it uses the resulting proof. If it fails (or if it simply does not have enough information) the tactic uses the `searchContext` tactic which searches the local and global contexts for a term which will solve the goal directly. If searching the context fails, it will use the `fail` tactic to report an appropriate, domain-specific error.

Aside — Parameters

To compute a proof that a value is an element of a vector, we need to be able to construct equality proofs between two elements of a type. Rather than pass around a function which does so, we parameterise a block of code over such a function:

```
params (eq:(x:a) -> (y:a) -> (Maybe (x = y))) {
  ...
}
isElem and isElemTac are parameterised over `eq`.

Example — Safe Access Control Lists

Consider a security policy which requires a user's ID to appear in an access control list before the user is given access to a resource.
Using a dependent type based approach [23], we could require a proof (using `Elem`) that a user is in a list of allowed users before being able to read a resource:

```haskell
allow : List User;
read_ok : (u:User) -> Resource ->
          Elem u allow -> IO Data;
```

To use `read_ok` we also need to provide a proof that the user is allowed access to the resource. Rather than provide this directly, we will leave a metavariable and provide a proof separately:

```haskell
answers = read_ok edwin exams ?;
```

Alternatively, we can invoke `isElemTac` directly, which will either construct a proof if possible, or search through the context for an existing proof if not, using the following syntax:

```haskell
answers = read_ok edwin exams
          [proof %intro; %decide isElemTac; %qed];
```

The `[proof ...]` syntax allows a tactic based proof to be inserted directly into a program. In practice, this is most useful when combined with syntax macros. We can define syntax for reading a resource, statically constructing a proof that access is permitted:

```haskell
syntax read u r
     = read_ok u r [proof %intro; %decide isElemTac; %qed];
```

Wherever `read u r` appears, the system tries to construct a proof of `Elem u r` using `isElemTac`, and reports a compile-time error if the tactic fails. An alternative syntax `[tryproof ...]` does not report an error if the tactic fails, but instead leaves the metavariable unsolved. This can be useful for implementing partial decision procedures.

Using this approach, we can construct functions with domain-specific correctness requirements, solved by domain-specific tactics, without the notational overhead of writing explicit metavariables and invoking tactics manually.

### 3. Extended Example: Binary Data Formats

In this section we present an extended example, an EDSL for describing, marshaling and unmarshaling binary data, which applies many of the techniques described in Section 2. We have chosen this example because it is a component of a real research project, rather than a contrived example, although we have omitted some of the details due to space restrictions. We are developing this embedded data description language as part of a project to describe and verify network protocols using dependent types [1].

Oury and Swierstra describe how a similar language could be implemented in principle [25]. In this section, we should how such a language can be implemented in practice, if it is to be implemented efficiently, and show some realistic data formats which it can encode.

#### 3.1 Primitive Binary Chunks

We are interested in parsing, manipulating and generating data to be transmitted across networks through verified network protocols. This involves manipulating data at the bit level. For example, implementing a TCP/IP stack would involve dealing with IP packets [29], the header of which is illustrated in Figure 3. To begin, we define `Chunk`, a universe of primitive components of binary data:

```haskell
data Chunk : Set where
    bit : (width: Int) -> so (width>0) -> Chunk
| CatString : Chunk
| LString : Int -> Chunk
| prop : (P:Prop) -> Chunk;
```

Primitive data can be a bit field, of a defined length greater than 0 (bit), a null-terminated C-style string (CatString), a string with an explicit length (LString) or a proposition about the data. Propositions are defined as follows, covering the basic logical connectives and relations, as well as a generic equality test `p_bool`:

```haskell
data Prop : Set where
    p_lt : Nat -> Nat -> Prop
| p_eq : Nat -> Nat -> Prop
| p_bool : Bool -> Prop
| p_and : Prop -> Prop -> Prop
| p_or : Prop -> Prop -> Prop;
```

Chunk and Prop each have a decoding to IDRIS types. In packet descriptions, we often need to work with numbers of a specific bit width (bit `n`), so we create a type based on machine integers, carrying a proof that the number is within required bounds:

```haskell
data Bounded : Int -> Set where
    BInt : (x:Int) -> so (x<i) -> Bounded i;
```

An instance of the `so` data type used above can only be constructed if its index is `True`. Effectively, this is a static proof that a dynamic check has been made:

```haskell
data so : Bool -> Set where
    oh : so True;
```

Rather than insert such proofs by hand, we construct a tactic which, like `isElemTac` in Section 2.8, constructs a proof if the bounds are statically known, or if a proof already exists in the context:

```haskell
isThatSo : (x:Bool) -> Tactic;
isThatSo x = Try (Fill oh)
    (Try SearchContext (Fail "That's not so!")));;
```

We define a syntax macro for bounded numbers which automatically invokes this tactic. Since the tactic only does a basic proof search, we use the `tryproof` construct, so that if the tactic fails then the programmer is free to construct a more complex proof by hand:

```haskell
syntax mk_so = [tryproof %intro; %decide isThatSo; %qed];
syntax bounded x = BInt x mk_so;
```

The decoding function for `Chunk` either gives a bounded number of the appropriate bit width (in the case of `bit`), a primitive `String` (in the case of `CString` or `LString`) or decodes a proposition.

```haskell
chunkTy : Chunk -> Set;
chunkTy (bit n) = Bounded (1 << n);
chunkTy (CString = String);
chunkTy (LString i) = String;
chunkTy (prop P) = propTy P;
```

Similarly, decoding propositions gives an appropriate IDRIS type:

```haskell
propTy : Prop -> Set;
propTy (p_lt x y) = LT x y;
propTy (p_eq x y) = x=y;
propTy (p_bool b) = so b;
propTy (p_and s t) = (propTy s & propTy t);
propTy (p_or s t) = Either (propTy s) (propTy t);
```

Given a proposition, we can also write a function which decides whether than proposition is true, suitable for use by the decide tactic, or for run-time checking. This proceeds structurally over a `Prop`:

```haskell
testProp : (p:Prop) -> Maybe (propTy p);
```
3.2 Packet Descriptions

Packet formats are composed of combinations of binary Chunks. We define a language of combinators, PacketLang, in Figure 4. This is an inductive-recursive definition — the data type is defined simultaneously with its decoding function. This is a standard technique [10], and is particularly effective for implementing embedded languages [20]. The decoding function mkTy is shown in Figure 5.

We overload do-notation to use BIND and CHUNK instead of (>>=) and return, and provide syntax macros for decluttering the syntax, in particular for automatically inserting proofs that bit widths are greater than 0:

```plaintext
syntax bits n = CHUNK (bit n mk_so);
syntax check n = CHUNK (prop (p_bool n));
syntax lstring n = CHUNK (lstring n);
... 
```

For example, a data format containing an 8-bit number, guaranteed to be greater than zero, followed by a string of that length would be described as follows, where value is a function which extracts an integer from a Bounded number:

```plaintext
string_format = do { len <- bits 8; 
check (value len > 0); 
lstring (value len); });
```

The decoding function for string_format yields nested dependent pairs, containing a length (with bounds proof), a proof that the length is greater than zero, and the string itself. We always write this type as mkTy string_format, and to work with it, we use a slightly different syntax x # y, which is more appropriate in this context in that we think of the # as a field separator:

```plaintext
syntax (#) x y = <| x , y |>; 
```

For greater flexibility, the language also includes a number of compound constructs. IF expressions allow alternative packet formats depending on a boolean value (e.g. computed from the form of earlier data):

```plaintext
IF : Bool -> PacketLang -> PacketLang -> PacketLang
| (//) : PacketLang -> PacketLang -> PacketLang
| LIST : PacketLang -> PacketLang
| LISTN : (n:Nat) -> PacketLang -> PacketLang
| BIND : (p:PacketLang) ->
  (mkTy p -> PacketLang) -> PacketLang;
```

We allow alternatives; a // b describes a packet format which can be either a or b:

```plaintext
(//) : PacketLang -> PacketLang -> PacketLang
```

---

**Figure 3. IP Header**

**Figure 4. Packet Descriptions**

```
mkTy : PacketLang -> Set;
data PacketLang : Set where
  CHUNK : (c:Chunk) -> PacketLang
| IF : Bool -> PacketLang -> PacketLang -> PacketLang
| (//) : PacketLang -> PacketLang -> PacketLang
| LIST : PacketLang -> PacketLang
| LISTN : (n:Nat) -> PacketLang -> PacketLang
| BIND : (p:PacketLang) ->
  (mkTy p -> PacketLang) -> PacketLang;
```

**Figure 5. Decoding Packets**

At their simplest, packet formats are a sequence of Chunks, combined with a BIND operator to allow the form of later chunks to depend on values in earlier chunks:

```
CHUNK : (c:Chunk) -> PacketLang;
BIND : (p:PacketLang) ->
  (mkTy p -> PacketLang) -> PacketLang;
```
Finally, there are two list constructs. LIST describes lists of arbitrary length, and LISTN describes lists of a specific length, perhaps computed from earlier data:

\[
\begin{align*}
\text{LIST} & : \text{PacketLang} \rightarrow \text{PacketLang} \\
\text{LISTN} & : (n:\text{Nat}) \rightarrow \text{PacketLang} \rightarrow \text{PacketLang}
\end{align*}
\]

### 3.3 Marshaling and Unmarshaling

Given a data format described in a PacketLang, we would like to be able to read and write concrete data in the format. Since there is no easy way to represent bit level data directly in IDRIS we represent it in C, as a pointer to a block of 32 bit words. In IDRIS, we wrap this pointer, and the length of the block, in the RawPkt type:

```hs
typedef word32* PACKET;  // C representation
data RawPkt = RPkt Ptr Int; -- Pointer to a PACKET
```

Packet descriptions can be sent and received over a network socket using the following functions implemented using the foreign function interface:

- **send**: Socket \(\rightarrow\) RawPkt \(\rightarrow\) IO ()
- **recv**: Socket \(\rightarrow\) IO (Maybe Recv)

Top level functions marshal and unmarshal then use packet descriptions to convert between the raw packet data and the high level representation of the packet type. We can think of these functions as “interpreters” for the packet language, whose semantics is to convert data from one form to another.

- **marshal**: (p:PacketLang) \(\rightarrow\) mkTy p \(\rightarrow\) RawPkt
- **unmarshal**: (p:PacketLang) \(\rightarrow\) RawPkt \(\rightarrow\) Maybe (mkTy p)

In order to implement marshal and unmarshal we will need to read and modify the contents of raw packets. This is achieved using foreign functions, which access packets by a location (as a bit offset) and a length in bits:

- **getField**: RawPkt \(\rightarrow\) Int \(\rightarrow\) (b:Int) \(\rightarrow\) IO (Maybe (BInt b))
- **setField**: RawPkt \(\rightarrow\) Int \(\rightarrow\) (b:Int) \(\rightarrow\) Bounded (1 \(\ll\) b) \(\rightarrow\) IO ()

Since we have given a length in bits, there is a clearly defined offset and length of the block, in the RawPkt type:

```hs
getField (RPkt pkt len) s b _ = if ((s\(\ll\)len) & (s\(\ll\)len)) then
  (Just (BInt (unsafePerformIO (getPacketBits (RPkt pkt len) s (e - 1)) (unsafeCoerce oh))))
else Nothing;
```

We have used two “unsafe” functions here: unsafePerformIO, since the C function does not modify anything, and so we can treat it as pure; and unsafeCoerce which converts the proof we can construct to the proof we need:

```hs
unsafeCoerce : {a:Set} \(\rightarrow\) {b:Set} \(\rightarrow\) a \(\rightarrow\) b;
```

### Aside — Contracts and Dynamic Checks

It is a little unsettling that we have used unsafeCoerce to create an efficient verified implementation. On the one hand, we argue that it is safe because we have access to the external C function’s implementation, which we can verify by hand; on the other hand, there ought to be a better way. For example, what if we change the C implementation and forget to update the IDRIS type?

We plan to improve this situation in the short term by extending FType to include constraints on foreign values, for example:

```hs
data FType = ... | FIntP (Int \rightarrow\) Bool);
```

The predicate `p` is effectively a contract which the foreign value must satisfy, similar to an assert in C. Like assert we would expect to be able to switch off checking when we are certain (either through extensive testing or verification of the external code) that the predicate will never return False.

### 3.4 A Simple Example

To show how packet formats work in practice, we give a simple packet format containing an IP address, followed by a list of strings with explicit lengths. An IP address is a sequence of 4 8-bit numbers:

```hs
IPAddress = do { len <- bits 8; bits 8; bits 8; bits 8; }
```

To represent strings with explicit lengths, we have an 8-bit number, followed by a string of exactly the given length. We will use a zero length to indicate the end of the list, so zero itself is an invalid length, which we will express as a constraint:

```hs
stringlist = LIST (do { len <- bits 8; check (value len > 0); 
LString (value len); })
```

The packet format is then an IP address, followed by a string list, followed by the end marker, which must be zero:
High level type conversion

To work with an instance of `mkTy` strings, we need to know how `mkTy` strings is concretely represented as a high level IDRIS type. In principle, it is enough to know that two elements `a` and `b` which appear in sequence in a `PacketLang` description will be concretely represented as `a # b`. For example, from a `mkTy` strings we can extract the components of the IP address, the list of strings, the end marker and the proof that the marker is zero as follows:

```haskell
readPkt : mkTy strings -> ...;
readPkt ((a # b # c # d) # xs # mark # prf) = ...;
```

In general, however, it would be much easier to work with a high level type rather than the generic types generated by `mkTy`. Therefore, we additionally define a high level type and conversion functions. These conversion functions will only be type correct if they do the appropriate validation when building the packet format:

```haskell
data StringData
  = SData (Int & Int & Int & Int) (List String);

readPkt : mkTy strings -> StringData;
writePkt : StringData -> Maybe (mkTy strings);
```

One fragment of the conversion converts a 4-tuple of integers to an IP address. The format specifies that the integers must be 8-bits, however, so we must construct a proof that the integers are within range. We write a dynamic checking function which, given a `Bool` constructs either a proof that it is false, or a proof that it is true:

```haskell
validCode : Int -> Int -> Bool;
validCode 0 x = x == 0;
validCode 3 x = x >=0 && x <= 13;
validCode 4 x = x == 0;
validCode 5 x = x >=0 && x <= 2;
validCode _ _ = False;
```

Using this, we convert integers to an IP address representation, shown in Figure 6. If the dynamic checks fail, and no proof can be constructed, the conversion fails. If they succeed, the bounded macro will retrieve the proofs generated by `choose` from the context.

```haskell
intIP : (Int & Int & Int & Int) ->
      Maybe (mkTy IPAddress);
intIP (a, b, c, d) with
    (choose (a < 256), choose (b < 256),
     choose (c < 256), choose (d < 256)) {  
      | (Right _, Right _, Right _, Right _) =
        Just (bounded a # bounded b #
              bounded c # bounded d);
      | _ = Nothing;
    }
```

Figure 6. IP Address Conversion

3.5 Network Packet Formats

More realistic packet formats follow the approach given above — describe the format and convert to an appropriate high level representation. Furthermore, in practice, while the `PacketLang` descriptions may be longer, they are rarely significantly more complex. We outline two: ICMP and IP headers.

ICMP

Figure 7 illustrates the layout of an ICMP (Internet Control Message Protocol) packet [28].

Figure 7. ICMP message

Valid codes, stored in bits 8–15, depend on the type, in bits 0–7. The 16-bit checksum depends on the content of the rest of the packet, and an invalid checksum renders the contents of the packet invalid. If we provide the checksum as part of the packet description, the unmarshaller will automatically verify the checksum, and the marshaller will automatically construct a valid checksum. The format of the message content also depends on the type and code. Figure 8 shows how this could be represented in `PacketLang`.

Figure 8. ICMP Packet Description

The ICMP type and code determine what form the message will take. Only certain combinations are valid:

```haskell
validCode : Int -> Int -> Bool;
validCode 0 x = x == 0;
validCode 3 x = x >=0 && x <= 13;
validCode 4 x = x == 0;
validCode 5 x = x >=0 && x <= 2;
... 
validCode _ _ = False;
```

We could also define valid codes using the IF construct, but this check, by pattern matching, is more convenient given the number of available codes. The message format also depends on the code and type, and is calculated by `ICMPformat` (the details of which we have omitted).

IP

Recall Figure 3, which gave an illustration of the IP header. Figure 9 gives code to represent an IP header as a `PacketLang`. Reading data from a concrete IP header is then simply a matter of calling `unmarshal IP_header`.

The `verify` function, as before, verifies that the header’s `checksum` field corresponds to a checksum calculated from the...
exploit this in particular to express dependencies and constraints of taking a generic programming approach in a dependently typed description languages [11] such as Packet.

Our data description language is related to monadic parsers such as Parsec [15] in that we provide a set of combinators for language descriptions. Oury and Swierstra take a similar approach using Agda [25], under the assumption that there are external functions for dealing with the details of bit processing. More generally, they propose dependently typed languages as a host for embedding domain specific languages with precise type systems. In this paper, we have taken this work further: not only do we embed a domain specific language for data description, we complete the implementation for processing real bit level data and show how the language can be used in practice.

Our embedded data description language is inspired by previous tools such as PacketTypes [22] and Mirage [18] (both specifically targeting network packet formats), and more general data description languages [11] such as PADS/ML [19]. One advantage of taking a generic programming approach in a dependently typed host is that we can use features of the host language directly. We exploit this in particular to express dependencies and constraints between data. Furthermore, the framework is extensible — we are not limited to the marshal and unmarshal functions. For example, we could extend the framework with pretty printing of packets or XML conversion, directed by a PacketLang format. Of course, as well as exploiting the features of the host language, we must also work with the limitations of the host language. Disadvantages of the generic approach, which we hope to address in future work, include the difficulty of producing good error messages and good error recovery in the parser.

5. Discussion

We have given an overview of the Idris programming language, including the most important language constructs and the distinctive features which make it suitable as a host language for a DSL. Our motivation is the need for verification of systems software — programs such as operating systems and network protocol implementations which are required for the correct operation of a computer system. Therefore it is important to consider not only how to verify software, but also how to do so efficiently, and how to inter-operate with concrete data as it is represented in the machine or on a network wire. The approach we have taken is to implement a generic data description language using a foreign function interface to access the concrete data. We use partial evaluation [5] to eliminate the abstraction overhead of the generic marshal and unmarshal functions. We have implemented some simple examples using this language, and tested them by marshaling and unmarshaling data over a real network connection.

Further Work

Our examples demonstrate the feasibility of using a dependently typed language to implement and verify systems software. To evaluate the approach fully, however, we will need significant examples with benchmarks, ideally implemented by systems programmers and network practitioners. There are some problems to be solved before we can apply our data description language in practice in this way:

1. The current implementation of Idris uses the Boehm-Demers-Weiser conservative garbage collector [2], which is fine for most applications, but may not be suitable for low level code which may need to run in very limited memory. A much simpler solution, of which we have a prototype implementation, would be to allocate a single pool of memory which is freed on return from a top level function. This region based approach is used to good effect in Hume [12].

2. In its current form PacketLang does not deal completely with bounds checking on integers. For example, we do not check that an integer is positive, or that it does not exceed an upper bound of 32 bits.

3. The error messages generated for incorrect EDSL programs can be hard to understand, particularly in cases where we have used syntactic sugar to make the EDSL programs more readable. To some extent we can mitigate the problem by generating more specific messages in domain-specific decision procedures but in general we will need to devise a mechanism for producing domain-specific error messages.

4. The path to adoption by systems programmers and network practitioners is not clear. Typically, they would use C or Java, and in order to be adopted any new approach would have to inter-operate with existing tools. A possible path to adoption would involve generating C headers for exported Idris functions, providing an API for accessing packet data. A good, simple, foreign function interface will be important here.
Systems software verification is vital — bugs in network and system software lead to security risks. We plan to use IDRIS and PacketLang formats to implement network applications, and generate benchmarks to compare our implementations with more conventional systems. The Domain Name Service (DNS) is a good candidate: it has a complex packet format with a built in string compression scheme which is hard to express in existing data description languages, and has been the source of serious bugs in DNS servers. Our hope is that language based verification techniques will help prevent such bugs in the future.

References


