

Idris, a language with dependent types — Extended Abstract

Edwin Brady

School of Computer Science,
University of St Andrews, St Andrews, Scotland.
Email: `eb@cs.st-andrews.ac.uk`.
Tel: +44-1334-463253, Fax: +44-1334-463278

Abstract. IDRIS is a functional programming language with full dependent types, built on top of IVOR, a theorem proving library for Haskell. The language provides a platform for practical programming with dependent types. Existing systems provide dependent types either through lightweight extensions, retaining a separation between types and values (e.g. GADTs in Haskell) or via full theorem proving requiring total correctness proofs (e.g. Agda, Epigram or Coq). In contrast, in IDRIS, we provide full dependent types while acknowledging that realistic programs require input/output, concurrency, and interfacing with foreign code, and furthermore that *totality* is not always necessary.

We make two main contributions: firstly, we give an implementation of an I/O monad with dependent types and show how, using dependent types, foreign functions can be incorporated directly into the type system without requiring any language extensions; secondly, we exploit the fact that IDRIS is built on top of a tactic based theorem prover to show how programming considerations can be separated from proof obligations, simplifying program development and readability.

1 Introduction

IDRIS is a functional programming language with full dependent types. It is similar to EPIGRAM [4] or AGDA [5], in that it supports dependent pattern matching. It is built on top of the IVOR [1] theorem proving library, and is a pure functional language with a syntax similar to Haskell with GADTs. The purpose of the language is to provide a platform for practical programming with dependent types.

1.1 Motivation

Several tools exist today for programming with dependent types, in various forms. These range from extensions to existing languages, such as Haskell's GADTs [?] and open type functions [?] to languages designed with dependent types from the ground up, such as Cayenne [?] and EPIGRAM [4]. Additionally, theorem provers such as AGDA [5] and COQ [?] are based on dependent types.

None of these systems, however, are suitable for writing realistic programs with full dependent types. The type system is either limited in some way, restricting the type level functions one can write and thus the properties which can be shown [?], or the focus is on total correctness, which, while clearly desirable, can make many simpler and less safety-critical programs more difficult to write than necessary.

In contrast, we are interested in the *practical* aspects of dependently typed programming — we require input and output, concurrency, and foreign function calls. We also acknowledge that at times, particularly in the early stages of software development, it may be necessary to relax some restrictions on the completeness of proofs. Application development and testing should not stall because of a missing lemma! Using our own implementation rather than an existing tool gives complete freedom to experiment with these abstractions and language features beyond the type system, such as I/O and concurrency. Additionally, although unrelated to the work we present in this paper, the core language is intended as an important step towards a fully dependently typed implementation of Hume [2].

1.2 Contributions

This paper introduces IDRIS, a language with dependent types, which compiles to executable code via C. The paper makes the following contributions:

1. We give an implementation of an I/O monad using dependent types, in the style of Hancock and Setzer [3], which compiles to C, and allows foreign functions (implemented in C) to be executed without requiring any extensions to the language or type system (Section 3).
2. We exploit the fact that IDRIS is built on top of a tactic based theorem prover, IVOR [1], to show how programming considerations and proof obligations can be neatly separated (Section 4).

Additionally, we demonstrate through a larger example, a verified implementation of binary arithmetic which we have previously implemented [?] in IVOR that a practical, realistic and efficient dependently typed programming language can be built with existing technology (Section 4.2).

2 Examples

3 I/O and Foreign Functions

We require input/output operations to be executable from within our language. To achieve this, we use Hancock and Setzer’s I/O model [3]. This involves implementing a Haskell-style I/O monad [?] by defining *commands*, representing externally executed I/O operations, and *responses* which give the type of the value returned by a given command. Our IO monad is then implemented in terms of these of commands and responses:

```

data IO : * → * where
  IOReturn : A → IO A
  | IODo : (c : Command) →
          (Response c → IO A) → IO A

```

Here, `IODo` takes a command, `c` and an I/O operation that transforms the response to that command into an `IO` value, and returns the result of applying the I/O operation to the command; and `Return` simply returns an action packaged as an `IO` value. To show how this works in practice, we can define simple commands and responses for reading and writing to standard input and output:

```

data Command : * where
  PutStr : String → Command
  | GetStr : Command
  | ...

```

The responses to these commands are a `String`, in the case that we have read a string, or the unit type if we have written a string.

```

Response : Command → *
Response (PutStr s) ↦ ()
Response GetStr ↦ String
...

```

The usual `bind` and `return` operations are simple to implement for this monad. We can now write the higher-level reading and writing operations:

```

getStr : IO String
getStr = IODo GetStr (λs:String. Return s)
putStr : String → IO Unit
putStr s = IODo (PutStr s) (λx:Unit. Return x)

```

Execution of an I/O program consists of evaluating it as normal and passing the result to an `EXECUTE` meta-operation. This is defined externally to the type theory and simply executes the I/O action at the head of the term, then evaluates and executes the continuation. Execution of the above simple string I/O language is defined by the following pseudo-Haskell code (for convenience, we will use Haskell-style `do` notation, with the obvious translation into our `IODo` and `Return` operations):

```

EXECUTE (IODo c r) ⇒ RUN c r
EXECUTE (Return v) ⇒ return v
RUN PutStr k ⇒ do str ← getLine
                EXECUTE (k str)
RUN (GetStr s) k ⇒ do putStr s
                EXECUTE (k unit)
...

```

In our compiler, which translates `IDRIS` to `C`, we implement this operation with a compiler pass which partially evaluates the `bind` operation and translates the commands into their `C` equivalents.

3.1 Foreign Function Calls

The IO monad as defined above requires us to decide which external operations are allowed, and to implement the appropriate RUN operation. While this does allow us to write realistic programs which communicate with the outside world, it lacks flexibility — there is no way to access arbitrary C libraries for, e.g., concurrency, graphics, or networking without modifying the implementation of RUN.

```
data   FType : * where
      FUnit : FType | Flnt : FType | FStr : FType | FPtr : FType
```

```
i_fType : FType → *
i_fType Flnt ↦ Int
i_fType FStr ↦ String
i_fType FUnit ↦ ()
```

```
data   ForeignFun : * where
      FFun : String → List FType → FType → ForeignFun
```

```
data   FArgList : List FType → * where
      fNil : FArgList Nil
      | fCons : (fx : i_fType x) → (fxs : FArgList xs) → FArgList (Cons x xs)
```

```
data   Command : * where
      ...
      | Foreign : (f : ForeignFun) → (args : FArgList (f.args f)) → Command
```

```
mkForeign : (f : ForeignFun) → mkFType f
mkForeign (FFun fn args ret) ↦ mkFDef fn args Nil fNil ret
```

4 Interactive Development

In writing dependently typed programs, we often find that expressing invariant properties in types means that a certain amount of theorem proving is necessary in order for a definition to typecheck. The following simple definition, for example, does not typecheck:

```
append : Vect A n → Vect A m → Vect A (m + n)
append nil ys ↦ ys
append (cons x xs) ys ↦ cons x (append xs ys)
```

The problem is in the second case; the left hand side and right hand side have different types:

```
append (cons x xs) ys : Vect A (m + (s n))
cons x (append xs ys) : Vect A (s (m + n))
```

Although these types are different, clearly they are equivalent. We often find that explicit coercions are needed, which prove equivalences between indices, to show that a definition is valid. Such coercions, however, are inconvenient for two reasons:

1. The required type for the coercion can be hard to find, without some assistance from the typechecker, and to write down such types can be tedious.
2. Inserting the coercion can make the code less readable.

4.1 Proposed definitions

In fact in this case, we can solve the problem simply by changing the type of **append** so that it returns $\text{Vect } A (n + m)$, but it will not always be so simple. Instead, we offer an alternative means of defining functions using the $?\mapsto$ operator. Using this operator we can give a proposed definition for a clause, and a name for the coercion we will define later which proves the equivalence between types:

```

append : Vect A n → Vect A m → Vect A (m + n)
append  nil    ys ↦ ys
append (cons x xs) ys ?↦ cons x (append xs ys)  [app_cons]

```

This generates a lemma **app_cons**. Before we can run **append**, we will need to give a definition for **app_cons**. The typechecker has generated the appropriate type for the lemma, and we can use the underlying theorem proving capabilities of IVOR to complete the definition:

```

app_cons : A → (Vect A n) → (Vect A m) → →
            (value : Vect A (s (m + n))) → Vect A (n + (s m))

```

The *value* argument is the proposed return value; we can now see its type, as generated by the typechecker. The return type is the type of the left hand side. The rôle of this lemma is to rewrite the type. All of the variables on the left hand side are passed as additional arguments to give the lemma access to the whole context. The real definition of **append** (internally, not required to be seen by the programmer) is then:

```

append : Vect A n → Vect A m → Vect A (m + n)
append  nil    ys ↦ ys
append (cons x xs) ys ↦ app_cons x xs ys (cons x (append xs ys))

```

For the purposes of testing, we provide a function which simply coerces one index to another, without checking. Obviously, any program which makes use of this cannot be trusted! However, it allows us to try proposed definitions and gain some intuition about how we may prove the required lemmas.

```

suspend_disbelief : (m : A) → (n : A) → m = n

```

4.2 Case study — binary arithmetic

Many dependently typed programs make use of natural numbers as indices to verify that size invariants are maintained. We gave a more complex example of this technique in [?], where we showed a binary adder to be correct by indexing binary numbers over their natural number equivalents. However, the requirement to use intermediate lemmas to show that the natural number invariants were indeed equivalent did cause the final program to be less readable than desired.

To recap briefly, we define bits as follows, indexed over their natural number representation:

```
data Bit : ℕ → ★ where 0 : Bit 0 | 1 : Bit (s 0)
```

We can then build on this to defined pairs of bits, then numbers and numbers with a carry flag, each of which carry the size of the corresponding ℕ in their type.

```
data BitPair : ℕ → ★ where
  bitpair : Bit c → Bit v → BitPair (v + (s (s 0)) × c)
```

```
data Number : ℕ → ℕ → ★ where
  none : Number 0 0
| bit : Bit b → Number n val →
  Number (s n) (((s (s 0))n × b) + val)
```

```
data NumCarry : ℕ → ℕ → ★ where
  numCarry : Bit c → Number n val →
  NumCarry n (((s (s 0))n × c) + val)
```

Assuming we have a means of adding bits, a full carry ripple adder can be implemented as follows, with the full types given. We add the most significant bits, given the result of adding the rest, as follows:

```
addNumberAux : Bit l → Bit r → NumCarry n val →
  NumCarry (s n) (((s (s 0))n × (l + r) + val)
addNumberAux x y (numCarry c num) ↦ msPair (addBit x y c) num
```

Then the adder is defined recursively:

```
addNumber : Number n l → Number n r → Bit carry →
  NumCarry n (carry + l + r)
addNumber none none c
  ↦ numCarry c none
addNumber (bit b1 num1) (bit b2 num2) c
  ↦ addNumberAux b1 b2 (addNumber num1 num2 c)
```

Unfortunately, and unsurprisingly, neither of these definitions typecheck, however convinced we might be that this is a correct implementation. In [?] we explain the reasons in detail, and show how to construct the required intermediate lemmas. Now in IDRIS, we can achieve this automatically, using the proposed definition syntax.

```

addNumberAux : Bit l → Bit r → NumCarry n val →
                NumCarry (s n) (((s (s 0))n) × (l + r) + val)
addNumberAux x y (numCarry c num)
                ?↦ msPair (addBit x y c) num    [num_aux]

addNumber : Number n l → Number n r → Bit carry →
            NumCarry n (carry + l + r)
addNumber none none c
            ?↦ numCarry c none    [none_case]
addNumber (bit b1 num1) (bit b2 num2) c
            ?↦ addNumberAux b1 b2 (addNumber num1 num2 c)    [bit_case]

```

We lose little readability from our original failed definition, and additionally are even in a position to test this function, by using the above **suspend_disbelief** function to fill in the proofs.

We would prefer, of course, to prove the lemmas completely. The typechecker has given us the lemmas **num_aux**, **none_case** and **bit_case**, which can be proved interactively using IVOR. We give the type of **num_aux** to illustrate:

```

num_aux : Bit c → Bit l → Bit r → NumCarry n val →
            (value : NumCarry (s n) (((s (s 0))n) * (r + y + c) + val)) →
            NumCarry (s n)((s (s 0))n × (l + r) + (((s (s 0))n) × c + val))

```

This type is fairly large (and so we are pleased to have the typechecker construct it for us!) but to prove the equivalence of indices requires straightforward algebraic manipulation. Such proofs could even be built automatically using a Presburger arithmetic solver. The main advantage of our approach is that it cleanly separates these proof requirements from the program, which aids both construction of the complete, verified program and, perhaps more importantly, readability of the program.

The full implementation of the binary adder (including a simple I/O interface) is available online, at <http://www.cs.st-andrews.ac.uk/~eb/drafts/binary.idr>.

5 Related Work

6 Conclusion

We have described the IDRIS programming languages and shown how to use it to implement a complex problem with dependent types. The language, while not approach the maturity of Haskell or other dependently typed systems such as AGDA, demonstrates that *existing technology* is sufficient to implement realistic and verified programs. Even a framework for foreign functions can be built neatly out of existing work, namely Hancock and Setzer's I/O trees [3]. It is particularly pleasing that foreign functions can be added without making *any* changes to the core language.

Any program with a sufficiently complex type will require a certain amount of theorem proving to typecheck. Our binary adder is an example of this. While this kind of program is expressible in weaker types systems such as that given by Haskell with open type families [?] the proof obligations become difficult to manage. It is therefore essential to provide some kind of interactive development tools along with any realistic dependently typed language. We have found the “proposed definition” syntax to reduce the burden on the programmer greatly; it allows a programmer to give the definition they believe to be correct, and even test it, and means that the *typechecker* generates the required lemma, rather than the programmer.

References

1. Lennart Augustsson. Cayenne - a language with dependent types. In *Proc. 1998 International Conf. on Functional Programming (ICFP '98)*, pages 239–250, 1998.
2. Edwin Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages 2006*, volume 4449 of *LNCS*. Springer, 2007.
3. Edwin Brady, Christoph Herrmann, and Kevin Hammond. Lightweight invariants with full dependent types. In *Draft Proceedings of Trends in Functional Programming 2008*, 2008.
4. Edwin Brady, James McKinna, and Kevin Hammond. Constructing correct circuits: Functional aspects of hardware specifications with dependent types. In *Trends in Functional Programming 2007*, 2007.
5. Coq Development Team. The Coq proof assistant — reference manual. <http://coq.inria.fr/>, 2001.
6. K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
7. Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proc. of 14th Ann. Conf. of EACSL, CSL'00, Fischbau, Germany, 21–26 Aug 2000*, volume 1862, pages 317–331. Springer-Verlag, Berlin, 2000.
8. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
9. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, 2007.
10. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. 2006 International Conf. on Functional Programming (ICFP 2006)*, 2006.
11. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.
12. Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP 2008*, 2008. To appear.