Cross-platform Compilers for Functional Languages

Research Paper

Edwin Brady

University of St Andrews, KY16 9SX, Scotland/UK,
ecb10@st-andrews.ac.uk

Abstract Modern software is often designed to run on a virtual machine, such as the JVM or .NET's CLR. Increasingly, even the web browser is considered a target platform with the Javascript engine as its virtual machine. The choice of programming language for a project is therefore restricted to the languages which target the desired platform. As a result, an important consideration for a programming language designer is the platform to be targetted. For a language to be truly cross-platform, it must not only support different operating systems (e.g. Windows, OSX and Linux) but it must also target different virtual machine environments such as JVM, .NET, Javascript and others. In this paper, I describe how this problem is addressed in the Idris programming language. The overall compilation process involves a number of intermediate representations and Idris exposes an interface to each of these representations, allowing back ends for different target platforms to decide which is most appropriate. I show how to use these representations to retarget Idris for multiple platforms, and further show how to build a generic foreign function interface supporting multiple platforms.

1 Introduction

Idris [7] is a functional programming language with dependent types, intended for *general purpose* programming. But what does "general purpose" mean in modern software development? Software runs in many diverse environments, not only on the desktop but also on mobile phones, tablets, as client code in web browsers, server code delivering web pages, and so on. These environments may run native code, virtual machine code (e.g. CLR or JVM), or they may interpret Javascript or PHP code. To describe a language as "general purpose", we ought to think how it can be adapted to work in all of these run time environments and more.

In this paper, I outline the overall compilation process for Idris, describing the intermediate representations (IRs) the code passes through before code is generated in a target language. There are several stages of IR, each stage removing some layer of abstraction, moving closer to low level code. Idris is implemented in Haskell, as a library, making it possible to expose each of these IRs to external programs. A code generator for Idris is a program which uses

Idris as a library to parse and type check Idris source files and generate an appropriate IR, then generates code for a specific target platform. I make the following specific contributions:

- I describe the various intermediate representations generated by Idris. These arise naturally during the default compilation path (via C), and since Idris is written as a library, they are exported for use by alternative code generators.
- I outline how an intermediate representation can be chosen to retarget Idris for a specific alternative back end, PHP.
- I present a generic foreign function interface (FFI) supporting calling external code, and calling Idris functions from external code. The FFI is implemented in Idris itself, taking full advantage of dependent types, with one primitive function and one primitive type treated specially by the compiler.

The FFI takes full advantage of dependent types to minimise the language extensions required to implement foreign functions. It could, however, be adapted to work as a primitive construct with similar features in a non dependently typed language. Otherwise, the techniques described in this paper would work equally well for implementing a cross-platform functional language with a different type system. Indeed, one thing we have observed is that a dependent type system changes very little about the techniques required for compiler implementation; in general, standard techniques work well. An exception is that we need to take greater care to erase irrelevant code [20].

2 Intermediate Representations

Idris programs are compiled through a series of intermediate representations, eventually resulting in executable code for some target platform. By default, Idris targets C, which is in turn compiled to native code. In this default compilation path, a program passes through the following representations:

- Idris itself. These programs include data type declarations, pattern matching definitions (with `where` clauses, `case` expressions, `do`-notation and other high level constructs), and type class and instance declarations.
- TT, a core language based on dependent type theory with fully explicit types [7]. The only top-level constructs are data type declarations and pattern matching definitions.
- $\mathsf{TT}_{\mathsf{case}}$, in which pattern matching is converted to simple `case` trees via a pattern match compiler [4].
- $\mathsf{IR}_{\mathsf{case}}$, in which all types are erased, as well as any values which are provably not used at run-time [20].
- $\mathsf{IR}_{\mathsf{lift}}$, in which there are no lambda bindings other than at the top level
- $\mathsf{IR}_{\mathsf{defunc}}$, in which *all* functions are fully applied. All functions are first-order, by defunctionalisation [18].

– $\mathsf{IR}_{\mathsf{ANF}}$, in which all functions are in *Applicative Normal Form (ANF)*, that is all arguments to functions are trivial [19]. By "trivial", we mean that evaluation of an argument must terminate immediately. In practice, this means it must be a variable or a constant.

Finally, the $\mathsf{IR}_{\mathsf{ANF}}$ form is simple enough to have an almost direct translation into C, via a bytecode format which deals with stack manipulation.

It is not the purpose of this paper to describe in detail the transformations between each of these phases. Indeed, the transformations are largely standard. However, each representation has its own characteristics, making it suitable for different code generators. Since all of the representations above are generated at some stage in the default compilation process, it is useful to expose all of them (using Idris as a *library*) in order to make them accessible by alternative code generators. In this section, therefore, I describe the intermediate representations focussing on $\mathsf{IR}_{\mathsf{case}}$ (of which $\mathsf{IR}_{\mathsf{lift}}$ is a special case), $\mathsf{IR}_{\mathsf{defunc}}$ and $\mathsf{IR}_{\mathsf{ANF}}$.

2.1 $\mathsf{IR}_{\mathsf{case}}$

Figure 1 shows the basic syntax of $\mathsf{IR}_{\mathsf{case}}$. In this figure, e stands for expressions, x for variables, $\mathbf{f}$ for global function names, i for constant literals, *alt* for case alternatives, *op* for primitive operators and d for the top level definitions. We use vector notation $\vec{x}$ to indicate a possibly empty sequence of x. Most of the syntax is standard; however, function applications in $\mathsf{IR}_{\mathsf{case}}$ are annotated with a flag *tail* which indicates whether the application should be treated as a tail call, and there is an expression form $\underline{\text{delay}}\ e$ which implements lazy evaluation, explicitly postponing the evaluation of the given expression. Constructors are annotated with an integer *tag*, which is guaranteed to be unique for each constructor of a single data type (but not necessarily unique between constructors of different data types). At this stage, all constructors are *saturated*.

A program in $\mathsf{IR}_{\mathsf{case}}$ consists of a collection of top level definitions and an expression to evaluate, i.e a pair $(\vec{d}, e)$. Top level definitions are either constructors (including their integer tag, and a record of their arity) or function definitions, consisting of a list of argument names and an expression.

As a convention, we use `typewriter` font for high level Idris programs, but when translated to any intermediate representation we will write **function** names in bold, *variable* names in italic and $\mathsf{constructor}$ names in sans serif.

2.2 Primitive Operators in $\mathsf{IR}_{\mathsf{case}}$

Primitive operators include arithmetic operators such as $+$, $-$, $\times$ and $\div$, comparison operators such as $<$, $\leq$, $>$ and $\geq$, and several convenient operators for working with other primitive types such as string manipulation. Code generators are not required to implement all primitive operators (indeed, some may be inappropriate or impossible for some targets) but are required to report an error if an unsupported operator is used.

$$
\begin{array}{lll|lll}
e ::= & x & \text{(variable)} & | & i & \text{(constant)} \\
| & \mathbf{f} & \text{(function name)} & | & e_{tail}\ \vec{e} & \text{(function)} \\
| & \mathsf{c}_i\ \vec{e} & \text{(constructor)} & | & \underline{\text{let}}\ x\ =\ e_1\ \underline{\text{in}}\ e_2 & \text{(let binding)} \\
| & \lambda\vec{x}.\ e & \text{(lambda bindings)} & | & \underline{\text{delay}}\ e & \text{(lazy evaluation)} \\
| & op\ \vec{e} & \text{(primitive)} & | & \underline{\text{case}}\ e\ \underline{\text{of}}\ \vec{alt} & \text{(case block)} \\
| & \underline{\text{error}}\ \langle string \rangle & \text{(abort)} & | & \Box & \text{(unused value)}
\end{array}
$$

$$
\begin{array}{rcll}
alt ::= & \mathsf{c}\ \vec{x} & \mapsto\ e & \text{(constructor case)} \\
| & i & \mapsto\ e & \text{(constant case)} \\
| & \_ & \mapsto\ e & \text{(default case)}
\end{array}
$$

$$
\begin{array}{rl}
d ::= & \mathsf{c}_i\ \langle arity \rangle\ \text{(Constructor declaration)} \\
| & x\ \vec{x}\ =\ e\ \text{(Function definition)}
\end{array}
$$

Figure 1. Expressions e and declarations d in $\mathsf{IR}_{\mathsf{case}}$

Primitive types are mostly conventional types covering integers, strings, and characters, and we will not go into any further detail on these. There is one other primitive type, WorldType and corresponding value World, used to implement I/O computations following a method similar to that used in GHC [16]. By this stage in the compilation process, I/O functions have been transformed such that any function written using the `IO` monad, returning a type...

```
IO a
```

...has been transformed so that in $\mathsf{IR}_{\mathsf{case}}$ it can be assumed to have the type...

$$\underline{\text{WorldType}}\ \rightarrow\ a$$

That is, it takes an extra argument indicating the current state of the world[1] and returns an unwrapped value. We say that it is assumed to have this type since $\mathsf{IR}_{\mathsf{case}}$ itself is untyped.

There are two primitive operators in $\mathsf{IR}_{\mathsf{case}}$ which use WorldType, to implement basic console I/O (again the types are for illustration only):

$$
\begin{array}{rl}
\mathit{WriteStr} & :\ \underline{\text{WorldType}} \rightarrow \underline{\text{String}} \rightarrow () \\
\mathit{ReadStr} & :\ \underline{\text{WorldType}} \rightarrow \underline{\text{String}}
\end{array}
$$

These are primitive, rather than implemented via a foreign function interface, in order to reduce the minimum requirement for getting a new code generator up and running. That is, there is no need for a code generator to implement a foreign function interface to be able to run simple interactive programs. Nevertheless, it is important to be able to support calls to foreign functions (though we will postpone the definition of exactly what we mean by "foreign" until later).

[1] In practice, this is just a dummy value.

2.3 Foreign Functions in $\mathsf{IR}_{\mathsf{case}}$

A realistic language implementation needs to interroperate with external libraries, to be able to call system services, communicate over a network, draw graphics, etc. To do so needs some kind of interface with *foreign* code. In Idris, we aim to do this in the most generic way possible, supporting multiple foreign languages rather than limiting to (for example) C functions.

$$
\begin{array}{rl}
e ::= & \ldots \\
\mid & \underline{\text{foreign}}\ fd_1\ fd_2\ \vec{fa} \text{ (foreign call)} \\
\\
fd ::= & \mathsf{c}\ \vec{fd} \quad \text{(Constructor application)} \\
\mid & \mathit{string} \text{ (String literal)} \\
fa ::= & (fd, e) \text{ (Foreign argument)}
\end{array}
$$

Figure 2. Extensions for foreign functions

Figure 2 shows an extension to $\mathsf{IR}_{\mathsf{case}}$ which supports foreign function calls. Here, *fd* is a *foreign descriptor*, which is a generic structure that can describe either the location of a foreign function (e.g. its name and the library where it is defined), its return type, or the type of its arguments. A foreign function call consists of a descriptor for the foreign function name, a descriptor for its return type, and a list of arguments each paired with a descriptor for the argument type. This gives a code generator enough information to call a foreign function in any language that specific generator supports. We will return to this in detail in Section 4.

2.4 $\mathsf{IR}_{\mathsf{defunc}}$

After $\mathsf{IR}_{\mathsf{case}}$, functions are lambda lifted into $\mathsf{IR}_{\mathsf{lift}}$. $\mathsf{IR}_{\mathsf{lift}}$ is exactly like $\mathsf{IR}_{\mathsf{case}}$, except that there are no λ bindings (hence we will not discuss $\mathsf{IR}_{\mathsf{lift}}$ further.) To achieve this, every λ is lifted to a named top-level function with variables in scope made explicit [17]. After this, the next step is to convert expressions to a *first order* form by defunctionalisation, resulting in the representation $\mathsf{IR}_{\mathsf{defunc}}$.

The syntax of $\mathsf{IR}_{\mathsf{defunc}}$, in practice, is that of $\mathsf{IR}_{\mathsf{lift}}$ without the $\underline{\text{delay}}$ expression form. However, there are additional semantic requirements that all functions are *fully applied*, and all applications are to explicitly named functions (i.e., not local variables.) As a result, we can be sure that *all* values are in canonical form (i.e. either a constructor form or a constant) and therefore there is no need for a run time system to implement closures.

This works by generating a data type with constructors representing functions which are not fully evaluated or fully applied, and generating functions **EVAL** and **APPLY**. **EVAL** evaluates a constructor representing an unevaluated function,

and **APPLY** applies a constructor representing a partially applied function to its next argument. For example, consider the following Idris function:

```
add : Int -> Int -> Int
add x y = x + y
```

There is a single data type, `Funs`, representing partially applied and unevaluated versions of all functions in an Idris program, which is constructed during the translation to $\mathsf{IR}_{\mathsf{defunc}}$. When a program using `add` is compiled, `Funs` will include representations of partially applied versions of `add` and a fully applied but unevaluated version of `add` (perhaps postponed due to lazy evaluation):

```
data Fns = ...
         | Add0
         | Add1 Int
         | Add2 Int Int
```

Correspondingly, **EVAL** and **APPLY** in the $\mathsf{IR}_{\mathsf{defunc}}$ representation will include the following cases:

EVAL *val* = case *val* of
  ...
  Add0 ↦ Add0
  Add1 *x* ↦ Add1 *x*
  Add2 *x y* ↦ add *x y*
  ...

APPLY *f a* = case *f* of
  ...
  Add0 ↦ Add1 *a*
  Add1 *x* ↦ add *x a*
  ...

In **EVAL**, we evaluate a representation of an unevaluated function. This is only possible if the representation is fully applied (i.e. `Add2`.) In **APPLY**, we add an extra argument to the representation, and if there are now enough arguments, run the function directly (i.e. adding an argument to `Add1`.)

A code generator can treat **EVAL** and **APPLY** like any ordinary function, and need not be aware that they are specially generated at all. This technique, defunctionalisation [18], greatly simplifies the task of writing a code generator and run time system since all of the work of managing higher order functions has been done by the compiler.

2.5 $\mathsf{IR}_{\mathsf{ANF}}$

The lowest level intermediate representation is $\mathsf{IR}_{\mathsf{ANF}}$ (Figure 3), which is in a variant of *Applicative Normal Form (ANF)*. In ANF, all function, constructor and primitive applications are to lists of *trivial* values, where a trivial value is

$$
\begin{array}{lllll}
e ::= x & \text{(variable)} & \mid i & \text{(constant)} \\
\quad \mid \mathbf{f}_{tail}\ \vec{v} & \text{(function)} & \mid \mathsf{c}_i\ \vec{v} & \text{(constructor)} \\
\quad \mid \underline{\text{let}}\ x\ =\ e_1\ \underline{\text{in}}\ e_2 & \text{(let binding)} & \mid op\ \vec{v} & \text{(primitive)} \\
\quad \mid \underline{\text{case}}\ e\ \underline{\text{of}}\ \vec{alt} & \text{(case block)} & \mid \underline{\text{constcase}}\ e\ \underline{\text{of}}\ \vec{calt} & \text{(case block)} \\
\quad \mid \underline{\text{error}}\ \langle string \rangle & \text{(abort)} & \mid \square & \text{(unused value)}
\end{array}
$$

$$
v ::= x \mid i \text{ (trivial values)}
$$

$$
\begin{array}{rl}
alt ::= & \mathsf{c}\ \vec{x}\ \mapsto\ e \text{ (constructor case)} \\
\mid & \_\ \mapsto\ e \text{ (default case)}
\end{array}
$$

$$
\begin{array}{rl}
calt \mid & i\ \mapsto\ e \text{ (constant case)} \\
\mid & \_\ \mapsto\ e \text{ (default case)}
\end{array}
$$

Figure 3. Expressions in $\mathsf{IR}_{\mathsf{ANF}}$

either a variable or a constant literal. Case expressions on constructors have also been explicitly separated from case expressions on constants.

The main advantage of $\mathsf{IR}_{\mathsf{ANF}}$, from a code generator author's point of view, is that all complex subexpressions have been lifted out into variables which, depending on the properties of the target machine, could be mapped to registers or stack locations. Furthermore, depending on the target, not all complex expression forms supported in higher level IRs may be supported on the target. Converting $\mathsf{IR}_{\mathsf{defunc}}$ to $\mathsf{IR}_{\mathsf{ANF}}$ is a relatively simple pass, simply replacing any complex subexpressions with a let bound variable and checking whether a case expression has constants or constructors in its branches.

2.6 Idris Code Generators

Idris is implemented in Haskell as a library providing support for (among other things) parsing and type checking of Idris source code, compiling via C, a read-eval-print loop, and generating the intermediate representations described in this section. As a result, a program can use the Idris library to preprocess or extend the language in various ways. One thing which is explicitly supported is plugging in alternative code generators.

When Idris successfully type checks a source file `program.idr`, it generates an intermediate file, `program.ibc`. This is a binary coded format for type checked Idris programs, containing everything needed to reload a program (including links to imported source files and external dependencies) without type checking again. A code generator, then, is a program which uses Idris as a library to load an `.ibc` file containing a collection of definitions, convert those definitions to an appropriate intermediate representation, then generates code for the desired target.

On the command line, Idris takes a flag `-codegen [generator]` to state which code generator should be used, if not the default. If told to use a code generator called `xyz` to build an executable `a.out` from an input file `Main.idr`, Idris will invoke the following command, after typechecking `Main.idr`:

```
idris-xyz Main.ibc -o a.out
```

It is therefore possible to extend Idris with arbitrarily many code generators by providing a command `idris-xyz` for some code generator `xyz`. A code generator built using Idris as a library has access to a `CodegenInfo` record provided by the library. This record contains information about a loaded `.ibc` file, including:

- Mappings from names to definitions in $\mathsf{IR}_{\mathsf{case}}$, $\mathsf{IR}_{\mathsf{lift}}$, $\mathsf{IR}_{\mathsf{defunc}}$ and $\mathsf{IR}_{\mathsf{ANF}}$, meaning that a code generator can choose whichever is most appropriate.
- Details about any external libraries to link.
- Details about any functions which should be exported, to be callable from foreign code. We will discuss this further in Section 4.5.

Essentially, therefore, the job of a code generator is to choose an appropriate intermediate representation, and map that representation to the desired target language. A target language with functional features (e.g. OCaml) may be easier to generate via $\mathsf{IR}_{\mathsf{case}}$, but a target language with imperative features may be easier to generate via $\mathsf{IR}_{\mathsf{defunc}}$ or even $\mathsf{IR}_{\mathsf{ANF}}$. In practice, $\mathsf{IR}_{\mathsf{ANF}}$ is sufficiently simple that any language supporting variables, a way of representing records, and function calls, is viable is a target language for compiled Idris programs.

3 Example: Targetting PHP

In this section, we briefly discuss how Idris can be translated directly into a simple target language, PHP, by choosing $\mathsf{IR}_{\mathsf{ANF}}$ as an intermediate representation. We choose PHP to illustrate this for the following reasons: at its core it is a very simple language, but nevertheless it has enough features to support running Idris code; furthermore it is very widely supported by web hosting providers even when no other way of creating dynamic web pages is available, therefore compiling to PHP provides a direct route to creating web applications in Idris which can be widely deployed.

After all, outside factors such as languages available on a host may affect the *run time environments* available to us, but they should not have to affect the *source language* available to us! The PHP code generator[2] itself is implemented in under 150 lines of Haskell code.

3.1 Representing Values

For any target, we need to consider how to represent Idris values at run-time. This includes primitive values such as strings and integers, as well as constructor

[2] Available from `https://github.com/edwinb/idris-php`

applications. Since closures have been removed by defunctionalisation, we do not need to consider how to represent these.

Since PHP is a dynamically typed language, we can represent primitive values directly and assume that the underlying system deals appropriately with managing these. Constructor applications can be represented as arrays, where the 0th element is the *tag*, and the following elements are the constructor arguments.

3.2 Compiling Pattern Matching

Pattern matching, by the time we reach $\mathsf{IR}_{\mathsf{ANF}}$, has been reduced to a tree of simple `case` expressions where a pattern is either a constant, a constructor applied to only variables, or a default "match anything" pattern. To implement `case` expressions, a back end needs to be able to inspect a constructor form to determine its tag, and project out its arguments into local variables.

In PHP, we can achieve this using the `switch` construct. To compile a case expression, `switch` on the first element of the array, and in each case bind local variables to the remaining arguments of the array.

3.3 Run Time Support

Most back ends will need some form of run time support, whether to implement primitive operations, initialisation, or garbage collection. For PHP, we provide a number of functions to implement primitive operations and handle errors:

```
function idris_error($str) { echo "$str\n"; exit(0); }
function idris_writeStr($str) { echo "$str\n"; }
function idris_append($l, $r) { return ($l . $r); }
```

3.4 Example

To illustrate briefly how a pattern matching function is translated to PHP via $\mathsf{IR}_{\mathsf{ANF}}$, consider a polymorphic function which appends two lists. In high level Idris notation, this is written as:

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

This translates to $\mathsf{IR}_{\mathsf{ANF}}$ as follows (making the constructor names explicit by replacing `[]` with nil, tag 0, and replacing `::` with cons, tag 1):

$$
\begin{array}{l}
\textbf{append}(\mathit{arg0}, \mathit{arg1}, \mathit{arg2}) \;\mapsto \\
\quad \underline{\text{case}}\ \mathit{arg1}\ \underline{\text{of}} \\
\qquad \mathsf{nil}_0 \;\mapsto\; \mathit{arg2} \\
\qquad \mathsf{cons}_1\ x\ \mathit{xs} \;\mapsto\; \underline{\text{let}}\ \mathit{loc3} \;=\; x\ \underline{\text{in}} \\
\qquad\qquad\qquad\qquad\quad \underline{\text{let}}\ \mathit{loc4} \;=\; \mathit{xs}\ \underline{\text{in}} \\
\qquad\qquad\qquad\qquad\quad \underline{\text{let}}\ \mathit{loc5} \;=\; \textbf{append}(\square, \mathit{loc4}, \mathit{arg2})\ \underline{\text{in}} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{cons}_1\ \mathit{loc3}\ \mathit{loc5}
\end{array}
$$

The first argument, *arg0*, corresponds to the type `a` in the original Idris code. This is an implicit argument, and is unused at run time, so is always replaced with $\Box$ in the $\mathsf{IR}_{\mathsf{ANF}}$ representation. Finally, this representation has a direct translation into PHP, using arrays to represent constructor application (Listing 1).

Listing 1. `append` in PHP

```
function append($arg0,$arg1,$arg2) {
  switch($arg1[0]) {
    case 0:
      return $arg2;
    case 1:
      $loc3 = $arg1[1]; $loc4 = $arg1[2];
      $loc5 = append(0,$loc4,$arg2);
      return array(1,$loc3,$loc5);
  }
}
```

4 Foreign Functions

A practical general purpose language needs to be able to call external code and interact with external libraries. Many high level languages (whether interpreted or compiled) have a built in interface to C, supporting conversions between high level values and C types, and allowing calling to and from C. C is treated here like a kind of portable assembler, or generic intermediate language. Idris takes a similar approach, with its default code generator.

However, C is not always appropriate. In aiming to be truly cross-platform, we must consider calling functions on alternative virtual machines such as the JVM where C is not an appropriate intermediate language. Ideally, the foreign function interface should support multiple virtual machines (including those we have not yet thought of!) To achieve this, the various intermediate representations support *generic* descriptions of foreign function calls, leaving the details to code generators. Idris itself provides high level data types for describing foreign function calls, as we describe in this section.

There is no special notation in Idris for describing foreign functions. For both calling external functions and exporting Idris functions for calling externally, everything is described using Idris functions and data types, some of which are treated specially by the compiler and converted to the foreign construct in $\mathsf{IR}_{\mathsf{case}}$.

4.1 Foreign Function Interfaces

A specific *foreign function interface* is described using the following Idris record:

```
record FFI : Type where
     MkFFI : (ffi_types : Type -> Type) ->
             (ffi_fn : Type) ->
             (ffi_data : Type) -> FFI
```

The fields of this record are `ffi_types`, which is a predicate describing which types can be passed to and from foreign functions, `ffi_fn` which is the type of function descriptors (e.g. for C, this is a `String` giving the function name) and `ffi_data` is the type of exported data descriptors (e.g. again for C, this is a `String` giving the name of a data type when exported to C.)

An Idris program calls foreign functions via the `foreign` function, defined in the Prelude:

```
foreign : (f : FFI) ->
          (fname : ffi_fn f) -> (ty : Type) ->
          {auto fty : FTy f [] ty} -> ty
```

From the programmer's point of view, this takes three arguments. Firstly, the FFI description under which the call will be made; secondly, the name (or more precisely the descriptor) of the function to be called; thirdly, the corresponding Idris type of the external function. The implicit argument `fty` is an automatically constructed proof that the given Idris type is valid in the given FFI (i.e. each component of the type satisfies the `ffi_types` predicate) using the following type:

```
data FTy : FFI -> List Type -> Type -> Type where
     FRet : ffi_types f t -> FTy f xs (IO' f t)
     FFun : ffi_types f s -> FTy f (s :: xs) t ->
            FTy f xs (s -> t)
```

While not strictly necessary for the proof, the list of types in the index helps maintain consistency between the FFI descriptor and the arguments, in the implementation of `foreign`; these details are beyond the scope of this paper.

Note that `FTy` requires that a foreign function returns a value in `IO' f t`. This is a version of the `IO` monad parameterised by the foreign functions it supports:

```
data IO' : (lang : FFI) -> Type -> Type
```

This is intended to prevent a function written for, say, interfacing with Javascript to attempt to interface with C. I will briefly discuss methods for writing `IO` programs which are generic in their FFI in Section 4.4.

4.2 Foreign calls to C

Values of primitive types can be passed between Idris and C, in which case the run time system will convert between the Idris and C representations. The `C_Types` predicate (Listing 2) describes the types which can be passed to and from C.

Listing 2. Foreign C Types

```
data C_Types : Type -> Type where
     C_Str   : C_Types String
     C_Float : C_Types Float
     C_Ptr   : C_Types Ptr
     C_Unit  : C_Types ()
     C_Any   : C_Types (Raw a)
     C_IntT  : C_IntTypes i -> C_Types i

data C_IntTypes : Type -> Type where
     C_IntNative : C_IntTypes Int
     C_IntChar   : C_IntTypes Char
     ... -- plus several fixed width types
```

The `Ptr` type is an Idris primitive representing `void*` pointers in C. Pointers can only be constructed in C and returned to Idris; there is no way to manipulate them directly in Idris code, other than to check for `NULL`. The `Raw` type allows arbitrary Idris values to be passed to C, without any marshaling. Finally, there are several integer types which are described in their own structure, `C_IntTypes`.

When a foreign call is translated into $\mathsf{IR}_{\mathsf{case}}$, it is the structure of this predicate which becomes the foreign descriptor. Functions in C are referred to directly by name. Hence the `FFI` declaration is written as follows:

```
FFI_C : FFI
FFI_C = MkFFI C_Types String String
```

The default `IO` monad in Idris supports foreign calls to C:

```
IO : Type -> Type
IO = IO' FFI_C
```

Example Consider the following C function, which opens a file:

```
FILE *fopen(char *filename, char *mode);
```

Treating `FILE*` as a generic pointer, we can write an Idris function which uses this to open a file (Listing 3). When translated to $\mathsf{IR}_{\mathsf{case}}$, the foreign call is:

$\underline{\text{foreign}}\ \mathsf{C\_Ptr}\ \texttt{"fopen"}\ (\mathsf{C\_String}, f)\ (\mathsf{C\_String}, m)$

Recall from Section 2.3 that a foreign call is annotated with descriptors for the return type (here $\mathsf{C\_Ptr}$) and for the argument types (here $\mathsf{C\_String}$), as well as a descriptor for the call itself (here `"fopen"`). A descriptor is either a constructor application, or a literal string, and can be used either for describing the location of an external function (using a `String` here), or describing the type of an argument or return value (using the `C_Types` predicate here.)

Listing 3. Opening a file via FFI

```
data File = Closed | Open Ptr

fopen : String -> String -> IO File
fopen f m = do fptr <- foreign FFI_C "fopen"
                          (String -> String -> IO Ptr)
                          f m
               if isNull fptr
                  then return Closed
                  else return (Open fptr)
```

4.3 Foreign calls to Javascript

Idris supports compilation to Javascript, as part of the default distribution. However, calling external Javascript code has several different considerations than calling external C code. Javascript libraries make extensive use of callbacks, in particular, therefore it is valuable to support passing *functions* to external Javascript. The predicate describing Javascript types is shown in Listing 4.

Listing 4. Foreign Javascript Types

```
data JS_Types : Type -> Type where
     JS_Str   : JS_Types String
     JS_Float : JS_Types Float
     JS_Unit  : JS_Types ()
     JS_FnT   : JS_FnTypes a -> JS_Types (JsFn a)
     JS_IntT  : JS_IntTypes i -> JS_Types i

data JS_IntTypes  : Type -> Type where
     JS_IntChar   : JS_IntTypes Char
     JS_IntNative : JS_IntTypes Int

data JsFn : Type -> Type where
     MkJsFn : (x : t) -> JsFn t

data JS_FnTypes : Type -> Type where
     JS_Fn     : JS_Types s -> JS_FnTypes t ->
                 JS_FnTypes (s -> t)
     JS_FnIO   : JS_Types t -> JS_FnTypes (IO' l t)
     JS_FnBase : JS_Types t -> JS_FnTypes t
```

As with C, we separate the integer types, using `JS_IntTypes`. We also separate function types. The Javascript FFI supports function types, provided that each component of the function type is itself valid. Programs which interface

with Javascript use a version of the `IO'` monad parameterised over a Javascript FFI descriptor:

```
FFI_JS : FFI
FFI_JS = MkFFI JS_Types String String

JS_IO : Type -> Type
JS_IO = IO' FFI_JS
```

4.4 Generic Foreign Calls

An advantage of our approach, parameterising an `IO'` monad over a FFI descriptor, is that it is a type error to attempt to call code in an unsupported language. Unfortunately, this also makes it difficult to write generic code. In a sense, this is hard to avoid; after all, the process for reading a file on one platform may be very different than on another. However, for application progammers, it would be nice to have a generic interface and let the compiler instantiate the appropriate FFI call. Let us assume we have defined file management functions for each of `FFI_C` and `FFI_JS`:

```
c_fopen : String -> String -> IO File
c_read  : File -> IO String
c_eof   : File -> IO Bool
c_close : File -> IO ()

js_fopen : String -> String -> JS_IO File
js_read  : File -> JS_IO String
js_eof   : File -> JS_IO Bool
js_close : File -> JS_IO ()
```

Listing 5. Reading a File

```
readFile : String -> IO (List String)
readFile f = do h <- c_fopen f "r"
                xs <- readLines h []
                c_close f
                return xs
  where
    readLines : File -> List String -> IO (List String)
    readLines h acc
      = do end <- c_eof h
           if end then return acc
              else do line <- c_read h
                      readLines h (acc ++ [line])
```

A function using this API to read a file, using `FFI_C`, is given in Listing 5. To use the same function in a program which targets Javascript would involve changing the return type to `JS_IO (List String)` and using the Javascript versions of the functions. Clearly, this is undesirable!

There are several potential ways of dealing with this, using the generic programming support provided by Idris. The simplest (but perhaps least elegant) method is to use type-directed overloading, e.g. using the name `fopen` for both of `c_fopen` and `js_open` and letting the machine disambiguate by type.

Type Classes A cleaner way would be to use type classes. Idris allows classes to be parameterised by *any* value, so we can define classes for packages of operations parameterised by an FFI:

```
class FileIO (f : FFI) where
  fopen : String -> String -> IO' f File
  read  : File -> IO' f String
  eof   : File -> IO' f Bool
  close : File -> IO' f ()

instance FileIO FFI_C where ...
instance FileIO FFI_JS where ...

readFile : FileIO f => String -> IO' f (List String)
```

Handlers of Algebraic Effects Cleaner still would be to define *algebraic effects* for packages of operations [8]. Algebraic effects allow the separation of descriptions of effects program from their implementation. Therefore, we can write a generic `readFile` program using a `FILE_IO` effect supporting the operations above, with the following type:

```
readFile : String -> Eff (List String) [FILE_IO]
```

Due to space limitations we will not go into detail of the effects system. Briefly, the `FILE_IO` effect consists of a signature (of type `FileIO`) containing the operations, and a resource (the file handle). The operations can be executed via the C and Javascript FFIs by defining *handlers* for the `FileIO` signature:

```
instance Handler FileIO IO where ...
instance Handler FileIO JS_IO where ...
```

4.5 Calling Idris from Foreign Code

To complete the foreign function interface, we must be able to call Idris functions from external code. Again, function exports are described entirely using Idris functions and data structures, then treated specially by a specific code generator.

Idris allows *monomorphic* data types to exported. This restriction is because there is no guarantee that a foreign language will support polymorphic types in the same way as Idris. Nevertheless, it is still useful to be able to export functions using user defined data types. Functions can be exported via an FFI provided that their argument and return types either satisfy the predicate given in the FFI, or are exported types. Foreign exports can be described using the `FFI_Export` data type in Listing 6.

Listing 6. Foreign Exports

```
data FFI_Export : (f : FFI) -> String ->
                  List (Type, ffi_data f) -> Type
     Data : (x : Type) -> (n : ffi_data f) ->
            FFI_Export f h ((x, n) :: xs) ->
            FFI_Export f h xs
     Fun : (fn : t) -> (n : ffi_fn f) ->
           {auto prf : FFI_Exportable f xs t} ->
           FFI_Export f h xs -> FFI_Export f h xs
     End : FFI_Export f h xs
```

A top level value of type `FFI_Export ffi fn []` is treated specially by the compiler, generating an export list. The list argument (initially `[]`) is used to ensure that any data type not supported by the `ffi` has been explicitly declared as exportable. This is the purpose of the `FFI_Exportable` argument (which we do not explain further here.)

A code generator which supports the given `ffi` should then use this export list to create an interface file `fn` which can be used by external code to call into Idris. For example, assume we have an Idris file containing the following:

```
append   : List Int -> List Int -> List Int
showList : List Int -> String
```

We can export these definitions for use in C programs by writing an export structure which specifically exports `List Int` as a type, as well as the `append` and `showList` functions:

```
exports : FFI_Export FFI_C "lists.h" []
exports = Data (List Int) "ListInt" $
          Fun append "append" $
          Fun showList "showList"
          End
```

When compiled with a code generator supporting `FFI_C`, this generates a header file `lists.h` containing versions of the functions accessible from C:

```
#include <idris_rts.h>
typedef VAL ListInt;
```

```
ListInt append(VM* vm, ListInt arg0, ListInt arg1);
char* showList(VM* vm, ListInt arg0);
```

The `VM*` argument is a pointer to virtual machine information, required by the run time system for memory management and garbage collection. Any C program which uses Idris functions as a library can create a virtual machine pointer with the `idris_vm` function provided by `idris_rts.h`, and delete it using `close_vm`:

```
VM* idris_vm();
void close_vm(VM* vm);
```

The present implementation uses a copying garbage collector, therefore there is no guarantee that any pointer returned by any of these functions will be stable after calling another Idris function using the same VM.

5 Related Work

The Idris compilation process uses several well-known techniques, including lambda lifting [17], defunctionalisation [18], pattern match compilation [4], and Applicative Normal Form [19]. There are several alternative approaches we could have taken, such as a variant of the STG machine used by GHC [15,13] or using continuation passing style (CPS) as an intermediate form [1]. Our goal, however, has been to keep the intermediate forms as simple and as first order as possible in order to simplify the task for code generator authors. The relationship between CPS and ANF is well known, and CPS has several practical benefits [11] including the ability to perform program transformations which are not possible on ANF such as full β-reduction. However, by the time compilation has reached the $\mathsf{IR}_{\mathsf{ANF}}$ representation, all optimising transformations have been applied in earlier stages.

A pure functional language ultimately needs to deal with impure and side-effecting code and interact with external libraries. The Idris approach to I/O follows that of GHC [16], with a special token representing the outside world, managed by an `IO` monad. Our approach to foreign functions differs, however. Instead of using annotations on functions and language pragmas, we note that it is possible to use Idris itself to describe types of foreign functions and export structures. An earlier version of Idris [6] used a similar method, but without the ability to distinguish between alternative foreign function interfaces.

The need for retargettable compilers is becoming increasingly clear, not only due to a desire for languages to run in the browser via Javascript, but also due to emerging compiler frameworks such as LLVM. GHC has an experimental code generator targetting LLVM [21], as does the MLton compiler for Standard ML [12]. There is also an implementations of Haskell which compiles to Javascript, GHCjs[3]. The work I describe in this paper aims to make any future effort for Idris much simpler, minimising the infrastructure any code generator needs.

[3] `https://github.com/ghcjs/ghcjs`

An earlier version of Idris used the Epic library [5] for code generation, but this was replaced due to a desire to experiment with alternative code generators. In future, it is possible that the current Idris compilation framework could be extracted in order to make Epic a flexible library for building retargettable compilers.

6 Conclusion

We have seen the various stages Idris uses to compile a high level program to executable code for a target machine, using lambda lifting, defunctionalisation and reduction to Applicative Normal Form (ANF). Using ANF, we have seen a brief example of how an Idris function can be translated to a simpler language, PHP. The PHP code generator full advantage of Idris as a library, using it to do the majority of the work. Exposing intermediate representations, and particularly exposing a completely first order representation, means it is easy to generate code for a new target. The PHP back end is less than 150 lines of Haskell code.

We have also seen how to implement a language independent foreign function interface which can be instantiated for a specific language by implementing a data type describing the types which can be passed between Idris and that language. To support a particular FFI, a code generator needs to convert the values and make a direct call.

Idris currently supports several target languages. The main distribution is able to target C and Javascript, there are known third party code generators which target LLVM, Java, Python, Ruby, the PHP code generator briefly described here, and an experimental code generator for the GAP algebra system [10]. In implementing the various code generators we have found using a defunctionalised intermediate representation to be very convenient. This essentially takes a difficult job away from a run time system, and moves the task to the compiler. It does, as a result, require whole program compilation rather than compilation by module. This has not yet caused any efficiency issues, and experience with other whole program compilers such as MLton[4] suggests that it should not be a serious problem in the long term. Nevertheless, we expect to be able to implement separate compilation to some extent, generating code for **EVAL** and **APPLY** at a final linking stage.

As well as the targets described above, there is work in progress to target the Erlang virtual machine [3]. The goal is to be able to interroperate with fault-tolerant communicating systems, and to be able to implement verified communicating systems directly in Idris. Since Erlang provides a state of the art fault-tolerant virtual machine [2], we believe it makes sense to use it as a target for distributed systems rather than implement a virtual machine from scratch.

There is also work in progress to implement a specific target for embedded systems. Rather than making one run time system suitable for all memory configurations, having a language which is easily retargettable means that we

[4] `http://mlton.org/`

can write a tailored run time system. For example, instead of a copying garbage collector, it may be more appropriate to use reference counting, or even try to avoid allocation altogether using linear or uniqueness types [9].

We have not discussed implementation of tail calls, except to note that IRs annotate function calls. Code generators are expected to respect these annotations, but this can be difficult if a target does not support tail calls natively, for example the JVM. The C code generator takes advantage of `gcc` optimisations to implement tail calls. A simple solution in many cases is to identify direct tail recursion and translate it into a loop, but specific solutions will depend on specific targets and whether it is considered acceptable to compromise on memory usage (and possibility of stack overflow) for efficiency or vice versa.

We might also expect difficult maintenance issues with multiple code generators. In practice, this does not seem to be a problem. The core type theory of Idris, TT has been stable for a long time. All high level language features are ultimately translated into TT, so new language features do not require modifications to any of the IRs. Furthermore, optimisations are typically performed at the level of TT or $\mathsf{IR}_{\mathsf{case}}$. As a result, code generators do not need modification when other features of Idris are added or changed, except where the addition has exposed a bug in the code generator itself.

We have not yet developed a comprehensive set of benchmarks, although this would clearly be desirable for several reasons. Firstly, benchmarks would allow code generator authors to compare their own implementations against others. Secondly, they would allow us to see more clearly the effect of high level program transformations. Finally, they would allow better comparison of alternative code generators for the *same* target, for example to establish which is a more efficient IR to work from. Rather than providing contrived benchmarks, we would like to collect a set of realistic Idris programs, similar to the nofib suite for Haskell [14].

Acknowledgements

My thanks to raichoo for his implementation of the Javascript code generator, as well as to the Scottish Informatics and Computer Science Alliance (SICSA) for financial support.

References

1. A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
2. J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.
3. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
4. L. Augustsson. Compiling Pattern Matching. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368—-381, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

5. E. Brady. Epic — a library for generating compilers. In *Trends in Functional Programming (TFP '11)*, 2011.
6. E. Brady. Idris — systems programming meets full dependent types. In *Programming Languages meets Program Verification (PLPV 2011)*, pages 43–54, 2011.
7. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013.
8. E. Brady. Resource-dependent algebraic effects. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming (TFP '14)*, volume 8843 of *LNCS*. Springer, 2014.
9. E. de Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness typing simplified. In O. Chitil, Z. Horváth, and V. Zsók, editors, *Implementation and Application of Functional Languages (IFL '07)*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007.
10. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.7.7*, 2015.
11. A. Kennedy. Compiling with continuations, continued. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, October 2007.
12. B. A. Leibig. An LLVM Back-end for MLton. Master's thesis, Rochester Institute of Technology, 2013.
13. S. Marlow and S. Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming, Snowbird*, pages 4–15, 2004.
14. W. Partain. The nofib benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Functional Programming*, Workshops in Computing. Springer Verlag, 1992.
15. S. Peyton Jones. Implementing lazy functional languages on stock hardware – the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.
16. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96, 2002.
17. S. Peyton Jones and D. Lester. A modular fully lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, May 1991.
18. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Proceedings of the ACM annual conference on ACM 72*, 2(30602):717–740, 1972.
19. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3-4):289–360, 1993.
20. M. Tejiščák and E. Brady. Practical erasure in dependently typed languages, 2015. Submitted.
21. D. A. Terei and M. M. Chakravarty. An LLVM backend for GHC. In *Proceedings of the Third ACM Haskell Symposium*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.