Epic — A Library for Generating Compilers

Edwin Brady

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

Abstract. Compilers for functional languages, whether strict or non-strict, typed or untyped, need to handle many of the same problems, for example thunks, lambda lifting, optimisation, garbage collection, and system interaction. Although implementation techniques are by now well understood, it remains difficult for a new functional language to exploit these techniques without either implementing a compiler from scratch, or attempting to fit the new language around another existing compiler. Epic is a compiled functional language which exposes functional compilation techniques to a language implementor, with a Haskell API. In this paper we describe Epic and outline how it may be used to implement a high level language compiler, illustrating our approach by implementing compilers for the λ-calculus and a dynamically typed graphics language.

1 Introduction

When implementing a new programming language, whether for research purposes or as a realistic general purpose language, we are inevitably faced with the problem of executing the language. Ideally, we would like execution to be as fast as possible, and exploit known techniques from many years of compiler research. However, it is difficult to make use of the existing available back ends for functional langauges, such as the STG [12, 15, 19] or ABC [18] machines. They may be too low level, they may make assumptions about the source language (e.g., its type system) or there may simply be no clearly defined API. As a result, experimental languages such as Agda [14] have resorted to generating Haskell, using `unsafeCoerce` to bypass the type system. Similarly, Cayenne [1] generated LML bypassing the type checker. This is not ideal for several reasons: we cannot expect to use the full power and optimisations of the underlying compiler, nor can we expect it to exploit any specific features of our new source language, such as the optimisation opportunities presented by rich dependent type systems [4].

Epic is a library which aims to provide the necessary features for implementing the back-end of a functional language — thunks, closures, algebraic data types, scope management, lambda lifting — without imposing *any* design choices on the high level language designer, other than encouraging a functional style. It provides *compiler combinators*, which guarantee that any output code will be syntactically correct and well-scoped. This gives a simple method for building a compiler for a new functional language, e.g., for experimentation with new type systems or new domain specific languages. In this paper, we describe Epic and its API using two example high level languages. More generally, we observe that:

1. Recent language and type system research has typically been based on extensions of existing languages, notably Haskell. While this makes implementation easier as it builds on an existing language, it discourages significant departures from the existing language (e.g., full dependent types). With Epic, we encourage more radical experiments by providing a standalone path to a realistic, efficient, language implementation.
2. A tool can become significantly more useful if it is embeddable in other systems. A language back end is no different — by providing an API for Epic, we make it more widely applicable. Haskell's expressiveness, particularly through type classes, makes it simple to provide an appropriate API for describing the core language.
3. Epic's small core and clearly defined API makes it a potential platform for experimentation with optimisations and new back ends. Indeed, we avoid implementation details in this paper. Several implementations are possible, perhaps targetting .NET or the JVM as well as native code.

Epic was originally written as a back end for Epigram [7] (the name[1] is short for "**Epi**gram **C**ompiler"). It is now used by Idris [5] and as an experimental back end for Agda. It is specifically designed for reuse by other source languages.

2 The Epic Language

Epic is based on the λ-calculus with some extensions. It supports primitives such as strings and integers, as well as tagged unions. There are additional control structures for specifying evaluation order, primitive loop constructs, and calling foreign functions. Foreign function calls are annotated with types, to assist with marshaling values between Epic and C, but otherwise there are no type annotations and there is no type checking — as Epic is intended as an intermediate language, it is assumed that the high level language has already performed any necessary type checking. The abstract syntax of the core language is given in Figure 1. We use de Bruijn telescope notation, $\vec{x}$, to denote a sequence of x. Variable names are represented by x, and i, b, f, c, and str represent integer, boolean, floating point, character and string literals respectively.

2.1 Definitions

An Epic program consists of a sequence of *untyped* function definitions, with zero or more arguments. The entry point is the function *main*, which takes no arguments. For example:

$$\begin{aligned} factorial(x) &= \underline{\text{if}}\ x == 0\ \underline{\text{then}}\ 1 \\ &\qquad\qquad \underline{\text{else}}\ x \times factorial(x-1) \\ main() &= putStrLn(intToString(factorial(10))) \end{aligned}$$

[1] Coined by James McKinna

$$
\begin{array}{lll}
p ::= \vec{def} & \text{(Epic program)} & def ::= x(\vec{x}) = t \quad \text{(Definition)} \\
\\
t ::= x & \text{(Variable)} & \mid \; t(\vec{t}) \quad \text{(Application)} \\
\mid \; \lambda x.\, t & \text{(Lambda binding)} & \mid \; \underline{\text{let}}\ x \ = \ t\ \underline{\text{in}}\ t \quad \text{(Let binding)} \\
\mid \; \mathsf{Con}\ i(\vec{t}) & \text{(Constructor)} & \mid \; t!i \quad \text{(Projection)} \\
\mid \; t\ op\ t & \text{(Infix operator)} & \mid \; \underline{\text{if}}\ t\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t \quad \text{(Conditional)} \\
\mid \; \underline{\text{case}}\ t\ \underline{\text{of}}\ \vec{alt} & \text{(Case expressions)} & \mid \; \underline{\text{lazy}}(t) \quad \text{(Lazy evaluation)} \\
\mid \; \underline{\text{effect}}(t) & \text{(Effectful term)} & \mid \; \underline{\text{while}}\ t\ t \quad \text{(While loops)} \\
\mid \; x := \ t\ \underline{\text{in}}\ t & \text{(Variable update)} & \mid \; \underline{\text{foreign}}\ T\ str\ (t\ \vec{:}\ T) \quad \text{(Foreign call)} \\
\mid \; \underline{\text{malloc}}\ t\ t & \text{(Allocation)} & \mid \; i \mid f \mid c \mid b \mid str \quad \text{(Constants)} \\
\\
alt ::= \mathsf{Con}\ i(\vec{x}) \ \mapsto \ t & \text{(Constructors)} & \mid \; i \ \mapsto \ t \quad \text{(Constants)} \\
\mid \; \underline{\text{default}} \ \mapsto \ t & \text{(Match anything)} & \\
\\
op ::= + \mid - \mid \times \mid / \mid == \mid < \mid \leq \mid > \mid \geq \mid << \mid >> & & \\
\\
T ::= \mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Bool} \mid \mathsf{Float} \mid \mathsf{String} & \text{(Primitives)} & \\
\mid \; \mathsf{Unit} & \text{(Unit type)} & \\
\mid \; \mathsf{Ptr} & \text{(Foreign pointers)} & \\
\mid \; \mathsf{Any} & \text{(Polymorphic type)} &
\end{array}
$$

Fig. 1. Epic syntax

The right hand side of a definition is an expression consisting of function applications, operators (arithmetic, comparison, and bit-shifting), bindings and control structures (some low level and imperative). Functions may be partially applied.

Values Values in an Epic program are either one of the primitives (an integer, floating point number, character, boolean or string) or a *tagged union*. Tagged unions are of the form $\mathsf{Con}\, i(t_1, \ldots, t_n)$, where i is the *tag* and the $\vec{t}$ are the *fields*. The name Con is to suggest "Constructor". For example, we could represent a list using tagged unions, with $\mathsf{Con}\ 0\ ()$ representing the empty list and $\mathsf{Con}\ 1\ (x,\ xs)$ representing a cons cell, where x is the element and xs is the tail of the list.

Tagged unions are inspected either using field projection ($t!i$ projects the ith field from a tagged union t) or by case analysis. E.g., to append two lists:

$$
\begin{array}{lll}
append(xs, ys) & = \underline{\text{case}}\ xs\ \underline{\text{of}} & \\
\quad \mathsf{Con}\ 0() & \mapsto & ys \\
\quad \mathsf{Con}\ 1(x, xs') & \mapsto & \mathsf{Con}\ 1(x, append(xs', ys))
\end{array}
$$

Evaluation strategy By default, expressions are evaluated eagerly (in applicative order), i.e. arguments to functions and tagged unions are evaluated immediately, left to right. Evaluation can instead be delayed using the lazy construct. An expression $\underline{\text{lazy}}(t)$ builds a thunk for the expression t which will not

be evaluated until it is required by one of: inspection in a case expression or the condition in an if statement; field projection; being passed to a foreign function; explicit evaluation with effect.An expression effect(t) evaluates the thunk t *without* updating the thunk with the result. This is to facilitate evaluation of side-effecting expressions.

Higher order functions Finally, expressions may contain λ and let bindings. Higher order functions such as *map* are also permitted:

$$\begin{aligned} map(f, xs) \;=\; & \underline{\text{case}}\ xs\ \underline{\text{of}} \\ & \quad \mathsf{Con}\ 0() \;\mapsto\; \mathsf{Con}\ 0() \\ & \quad \mathsf{Con}\ 1(x, xs') \;\mapsto\; \mathsf{Con}\ 1(f(x), map(f, xs')) \end{aligned}$$

$$\begin{aligned} evens(n) \;=\; & \underline{\text{let}}\ nums \;=\; take(n, countFrom(1))\ \underline{\text{in}} \\ & \quad map(\lambda x.\ x \;\times\; 2, nums) \end{aligned}$$

2.2 Foreign Functions

Most programs eventually need to interact with the operating system. Epic provides a lightweight foreign function interface which allows interaction with external C code. Since Epic does no type checking or inference, a foreign call requires the argument and return types to be given explicitly. e.g. the C function:

```
double sin(double x);
```

We can call this function from Epic by giving the C name, the return type (an Epic Float, which corresponds to a C `double`) and the argument type (also an Epic Float).

$$sin(x) \;=\; \underline{\text{foreign}}\ \mathsf{Float}\ \text{"}\texttt{sin}\text{"}\ (x \;:\; \mathsf{Float})$$

2.3 Low Level Features

Epic emphasises control over safety, and therefore provides some low level features. A high level language may wish to use these features in some performance critical contexts, whether for sequencing side effects, implementing optimisations, or to provide run-time support code. Epic allows sequencing, while loops and variable update, and provides a malloc construct for memory allocation. The behaviour of malloc $n\,t$ is to create a fixed pool of n bytes, and allocate only from this pool when evaluating t. Due to space restrictions we will not discuss these constructs further.

2.4 Haskell API

The primary interface to Epic is through a Haskell API which is used to build expressions and programs with higher order abstract syntax (HOAS) [17]. Implementing a compiler for a high level language then involves converting the abstract syntax of a high level program into an Epic program, through these "compiler combinators", and implementing any run-time support as Epic functions.

Programs and expressions The API allows the building of Epic programs with an Embedded Domain Specific Language (EDSL) style interface, i.e. we try to exploit Haskell's syntax as far as possible. There are several possible representations of Epic expressions. `Expr` is the internal abstract representation, and `Term` is a representation which carries a name supply. We have a type class `EpicExpr` which provides a function `term` for building a concrete expression using a name supply:

```
type Term = State NextName Expr
class EpicExpr e where
    term :: e -> Term
```

There are straightforward instances of `EpicExpr` for the internal representations `Expr` and `Term`. There is also an instance for `String`, which parses concrete syntax, which is beyond the scope of this paper. More interestingly, we can build an instance of the type class which allows Haskell functions to be used to build Epic functions. This means we can use Haskell names for Epic references, and not need to worry about scoping or ambiguous name choices.

```
instance EpicExpr e => EpicExpr (Expr -> e) where
```

Alternatively, function arguments can be given explicit names, constructed with `name`. A reference to a name is built with `ref`, and `fn` composes `ref` and `name`.

```
name :: String -> Name
ref  :: Name   -> Term
fn   :: String -> Term
```

We provide an instance of `EpicExpr` to allow a user to give names explicitly. This may be desirable when the abstract syntax for the high level language we are compiling has explicit names.

```
instance EpicExpr e => EpicExpr ([Name], e) where
```

For example, the identity function can be built in either of the following ways:

```
id1, id2 :: Term
id1 = term (\ x -> x)
id2 = term ([name "x"], fn "x")
```

Both forms, using Haskell functions or explicit names, can be mixed freely in an expression. A program is a collection of named Epic expressions built using the `EpicExpr` class:

```
type Program = [EpicDecl]
data EpicDecl = forall e. EpicExpr e => EpicFn Name e
```

The library provides a number of built-in definitions for some common operations such as outputting strings and converting data types:

```
basic_defs :: [EpicDecl]
```

In this paper we use *putStr* and *putStrLn* for outputting strings, *append* for concatenating strings, and *intToString* for integer to string conversion. We can compile a collection of definitions to an executable, or simply execute them directly. Execution begins with the function called `"main"` — Epic reports an error if this function is not defined:

```
compile :: Program -> FilePath -> IO ()
run     :: Program -> IO ()
```

Building expressions We have seen how to build λ bindings with the `EpicExpr` class, using either Haskell's λ or pairing explicitly bound names with their scope. We now add further sub-expressions. The general form of functions which build expressions is to create a `Term`, i.e. an expression which manages its own name supply by combining arbitrary Epic expressions, i.e. instances of `EpicExpr`. For example, to apply a function to an argument, we provide an `EpicExpr` for the function and the argument:

```
infixl 5 @@
(@@) :: (EpicExpr f, EpicExpr a) => f -> a -> Term
```

Since `Term` itself is an instance of `EpicExpr`, we can apply a function to several arguments through nested applications of `@@`, which associates to the left as with normal Haskell function application. We have several arithmetic operators, including arithmetic, comparison and bitwise operators, e.g.:

```
op_ :: (EpicExpr a, EpicExpr b) => Op -> a -> b -> Term
plus_, minus_, times_, divide_ :: Op
```

We follow the convention that Epic keywords and primitive operators are represented by a Haskell function with an underscore suffix. This convention arises because we cannot use Haskell keywords such as `if`, `let` and `case` as function names. For consistency, we have extended the convention to all functions and operators. `if...then...else` expressions are built using the `if_` function:

```
if_ :: (EpicExpr a, EpicExpr t, EpicExpr e) => a -> t -> e -> Term
```

For let bindings, we can either use HOAS or bind an explicit name. To achieve this we implement a type class and instances which support both:

```
class LetExpr e where
    let_ :: EpicExpr val => val -> e -> Term

instance EpicExpr sc => LetExpr (Name, sc)
instance                LetExpr (Expr -> Term)
```

To build a constructor form, we apply a constructor with an integer *tag* to its arguments. We build a constructor using the `con_` function, and provide a shorthand `tuple_` for the common case where the tag can be ignored — as the name suggests, this happens when building tuples and records:

```
con_   :: Int -> Term
tuple_ :: Term
```

Case analysis To inspect constructor forms, or to deconstruct tuples, we use case expressions. A case expression chooses one of the alternative executions path depending on the value of the scrutinee, which can be any Epic expression:

```
case_ :: EpicExpr e => e -> [Case] -> Term
```

We leave the definition of `Case` abstract (although it is simply an Epic expression carrying a name supply) and provide an interface for building case branches. The scrutinee is matched against each branch, in order. To match against a constructor form, we use the same trick as we did for λ-bindings, either allowing Haskell to manage the scope of constructor arguments, or giving names explicitly to arguments, or a mixture. For convenience, we make `Expr` and `Term` instances to allow empty argument lists.

```
class Alternative e where
    mkAlt :: Tag -> e -> Case
instance Alternative Expr
instance Alternative Term
instance Alternative e => Alternative (Expr -> e)
instance Alternative e => Alternative ([Name], e)
```

We can build case alternatives for constructor forms (matching a specific tag), tuples, or integer constants (matching a specific constant), and a default case if all other alternatives fail to match. In the following, `e` is an expression which gives the argument bindings, if any, and the right hand side of the match.

```
con         :: Alternative e => Int -> e -> Case
tuple       :: Alternative e =>        e -> Case
constcase   :: EpicExpr e    => Int -> e -> Case
defaultcase :: EpicExpr e    =>        e -> Case
```

3 Example — Compiling the λ-Calculus

In this section we present a compiler for the untyped λ-calculus using HOAS, showing the fundamental features of Epic required to build a complete compiler.

3.1 Representation

Our example is an implementation of the untyped λ-calculus, plus primitive integers and strings, and arithmetic and string operators. The Haskell representation uses higher order abstract syntax (HOAS). We also include global references (`Ref`) which refer to top level functions, function application (`App`), constants (`Const`) and binary operators (`Op`):

```
data Lang = Lam (Lang -> Lang)
          | Ref Name
          | App Lang Lang
          | Const Const
          | Op Infix Lang Lang
```

Constants can be either integers or strings:

```
data Const = CInt Int | CStr String
```

There are infix operators for arithmetic (`Plus`, `Minus`, `Times` and `Divide`), string manipulation (`Append`) and comparison (`Eq`, `Lt` and `Gt`). The comparison operators return an integer — zero if the comparison is true, non-zero otherwise:

```
data Infix = Plus | Minus | Times | Divide | Append | Eq | Lt | Gt
```

A complete program consists of a collection of named `Lang` definitions:

```
type Defs = [(Name, Lang)]
```

3.2 Compilation

Our aim is to convert a collection of `Defs` into an executable, using the `compile` or `run` function from the Epic API. Given an Epic `Program`, `compile` will generate an executable, and `run` will generate an executable then run it. Recall that a program is a collection of named Epic declarations:

```
data EpicDecl = forall e. EpicExpr e => EpicFn Name e
type Program = [EpicDecl]
```

Our goal is to convert a `Lang` definition into something which is an instance of `EpicExpr`. We use `Term`, which is an Epic expression which carries a name supply. Most of the term construction functions in the Epic API return a `Term`.

```
build :: Lang -> Term
```

The full implementation of `build` is given in Figure 2. In general, this is a straightforward traversal of the `Lang` program, converting `Lang` constants to Epic constants, `Lang` application to Epic application, and `Lang` operators to the appropriate built-in Epic operators.

```
build :: Lang -> Term
build (Lam f)          = term (\x -> build (f (EpicRef x)))
build (EpicRef x)      = term x
build (Ref n)          = ref n
build (App f a)        = build f @@ build a
build (Const (CInt x)) = int x
build (Const (CStr x)) = str x
build (Op Append l r)  = fn "append" @@ build l @@ build r
build (Op op l r)      = op_ (eOp op) (build l) (build r)
    where eOp Plus   = plus_
          eOp Minus  = minus_
          ...
```

Fig. 2. Compiling Untyped λ-calculus

Using HOAS has the advantage that Haskell can manage scoping, but the disadvantage that it is not straightforward to convert the abstract syntax into another form. The Epic API also allows scope management using HOAS, so we need to convert a function where the bound name refers to a `Lang` value into a function where the bound name refers to an Epic value. The easiest solution is to extend the `Lang` datatype with an Epic reference:

```
data Lang = ...
          | EpicRef Expr

build (Lam f) = term (\x -> build (f (EpicRef x)))
```

To convert a `Lang` function to an Epic function, we build an Epic function in which we apply the `Lang` function to the Epic reference for its argument. Every reference to a name in `Lang` is converted to the equivalent reference to the name in Epic. Although it seems undesirable to extend `Lang` in this way, this solution is simple to implement and preserves the desirable feature that Haskell manages scope. Compiling string append uses a built in function provided by the Epic interface in `basic_defs`:

```
build (Op Append l r) = fn "append" @@ build l @@ build r
```

Given `build`, we can translate a collection of HOAS definitions into an Epic program, add the built-in Epic definitions and execute it directly. Recall that there must be a *main* function or Epic will report an error — we therefore add a main function which prints the value of an integer expression given at compile time.

```
main_ exp = App (Ref (name "putStrLn"))
                (App (Ref (name "intToString")) exp)

mkProgram :: Defs -> Lang -> Program
mkProgram ds exp = basic_defs ++
                   map (\ (n, d) -> EpicFn n (build d)) ds ++
                   [(name "main", main_ exp)]

execute :: Defs -> Lang -> IO ()
execute p exp = run (mkProgram p exp)
```

Alternatively, we can generate an executable. Again, the entry point is the Epic function *main*:

```
comp :: Defs -> Lang -> IO ()
comp p exp = compile "a.out" (mkProgram p exp)
```

This is a compiler for a very simple language, but a compiler for any more complex language follows the same pattern: convert the abstract syntax for each named definition into a named Epic `Term`, add any required primitives (we have just used `basic_defs` here), and pass the collection of definitions to `run` or `compile`.

4 Atuin — A Dynamically Typed Graphics Language

In this section we present a more detailed example language, Atuin[2], and outline how to use Epic to implement a compiler for it. Atuin is a simple imperative language with higher order procedures and dynamic type checking, with primitive operations implementing turtle graphics. The following example illustrates the basic features of the language. The procedure `repeat` executes a code block a given number of times:

```
repeat(num, block) {
  if num > 0 {
     eval block
     repeat(num-1, block)
  }
}
```

Using `repeat`, `polygon` draws a polygon with the given number of sides, a size and a colour:

```
polygon(sides, size, col) {
  if sides > 2 {
    colour col
    angle = 360/sides
    repeat(sides, {
      forward size
      right angle
    })
  }
}
```

Programs consist of a number of procedure definitions, one of which must be called `main` and take no arguments:

```
main() {
  polygon(10,25,red)
}
```

4.1 Abstract Syntax

The abstract syntax of Atuin is defined by algebraic data types constructed by a Happy-generated parser. Constants can be one of four types: integers, characters, booleans and colours:

```
data Const = MkInt Int   | MkChar Char
           | MkBool Bool | MkCol Colour

data Colour = Black | Red | Green | Blue | ...
```

Atuin is an imperative language, consisting of sequences of commands applied to expressions. We define expressions (`Exp`) and procedures (`Turtle`) mutually.

[2] `http://hackage.haskell.org/package/atuin`

Expressions can be constants or variables, and combined by infix operators. Expressions can include code blocks to pass to higher order procedures.

```
data Exp = Infix Op Exp Exp | Var Id
         | Const Const      | Block Turtle
data Op = Plus | Minus | Times  | Divide | ...
```

Procedures define sequences of potentially side-effecting turtle operations. There can be procedure calls, turtle commands, and some simple control structures. `Pass` defines an empty code block:

```
data Turtle = Call Id [Exp]     | Turtle Command
            | Seq Turtle Turtle | If Exp Turtle Turtle
            | Let Id Exp Turtle | Eval Exp
            | Pass
```

The turtle can be moved forward, turned left or right, or given a different pen colour. The pen can also be raised, to allow the turtle to move without drawing.

```
data Command = Fd Exp     | RightT Exp | LeftT Exp
             | Colour Exp | PenUp      | PenDown
```

As with the λ-calculus compiler in Section 3, a complete program consists of a collection of definitions, where definitions include a list of formal parameters and the program definition:

```
type Proc = ([Id], Turtle)
type Defs = [(Id,  Proc)]
```

4.2 Compiling

While Atuin is a different kind of language from the λ-calculus, with complicating factors such as a global state (the turtle), imperative features, and dynamic type checking, the process of constructing a compiler follows the same general recipe, i.e. define primitive operations as Epic functions, then convert each Atuin definition into the corresponding Epic definition.

Compiling Primitives The first step is to define primitive operations as Epic functions. The language is dynamically typed, therefore we will need primitive operations to check dynamically that they are operating on values of the correct type. We define functions which construct Epic code for building values, effectively using a single algebraic datatype to capture all possible run-time values (i.e. values are "uni-typed" [20]).

```
mkint  i = con_ 0 @@ i
mkchar c = con_ 1 @@ c
mkbool b = con_ 2 @@ b
mkcol  c = con_ 3 @@ c
```

Correspondingly, we can extract the concrete values safely from this structure, checking that the value is the required type, e.g.

```
getInt x  = case_ x [con 0 (\ (x :: Expr) -> x),
                     defaultcase (error_ "Not an Int")]
```

Similarly, `getChar`, `getBool` and `getCol` check and extract values of the appropriate type. Using these, it is simple to define primitive arithmetic operations which check that they are operating on the correct type, and report an error if not.

```
primPlus   x y = mkint $ op_ plus_   (getInt x) (getInt y)
primMinus  x y = mkint $ op_ minus_  (getInt x) (getInt y)
primTimes  x y = mkint $ op_ times_  (getInt x) (getInt y)
primDivide x y = mkint $ op_ divide_ (getInt x) (getInt y)
```

Graphics Operations We use the Simple DirectMedia Layer[3] (SDL) to implement graphics operations. We implement C functions to interact with SDL, and use Epic's foreign function interface to call these functions. For example:

```
void* startSDL(int x, int y);
void  drawLine(void* surf, int x, int y, int ex, int ey,
                           int r, int g, int b, int a);
```

The `startSDL` function opens a window with the given dimensions, and returns a pointer to a *surface* on which we can draw; `drawLine` draws a line on a surface, between the given locations, and in the given colour, specified as red, green, blue and alpha channels.

We represent colours as a 4-tuple (r, g, b, a). Drawing a line in Epic involves extracting the red, green, blue and alpha components from this tuple, then calling the C `drawLine` function. To make a foreign function call, we use `foreign_`, giving the C function name and explicit types for each argument so that Epic will know how to convert from internal values to C values:

```
drawLine :: Expr -> Expr -> Expr -> Expr -> Expr -> Expr -> Term
drawLine surf x y ex ey col
    = case_ (rgba col)
        [tuple (\ r g b a ->
           foreign_ tyUnit "drawLine"
             [(surf, tyPtr),
              (x, tyInt), (y, tyInt), (ex, tyInt), (ey, tyInt),
              (r, tyInt), (g, tyInt), (b, tyInt), (a, tyInt)]) ]
```

The turtle state is a tuple (s, x, y, d, c, p) where s is a pointer to the SDL surface, $(x,\ y)$ gives the turtle's location, d gives its direction, c gives the colour and p gives the pen state (a boolean, false for up and true for down). Note that this state is not accessible by Atuin programs, so we do not dynamically check each component. To implement the `forward` operation, for example, we take the current state, update the position according to the distance given and the current direction, and if the pen is down, draws a line from the old position to the new position.

[3] `http://libsdl.org/`

```
forward :: Expr -> Expr -> Term
forward st dist = case_ st
  [tuple (\ (surf :: Expr) (x :: Expr) (y :: Expr)
            (dir :: Expr) (col :: Expr) (pen :: Expr) ->
    let_ (op_ plus_ x (op_ times_ (getInt dist) (esin dir)))
      (\x' -> let_ (op_ plus_ y (op_ timesF_ (getInt dist) (ecos dir)))
      (\y' -> if_ pen (fn "drawLine" @@ surf @@ x @@ y
                                    @@ x' @@ y' @@ col) unit_ +>
              tuple_ @@ surf @@ x' @@ y' @@ dir @@ col @@ pen)))]
```

Here we have applied `getInt`, `esin` and `ecos` as Haskell functions, so they will be inlined in the resulting Epic code. In contrast, `drawLine` is applied as a separately defined Epic function, using Epic's application operator (`@@`).

Compiling Programs Programs return an updated turtle state, and possibly perform side-effects such as drawing. An Atuin definition with arguments $a_1 \ldots a_n$ is translated to an Epic function with a type of the following form:

$$f \ : \ State \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow State$$

To compile a complete program, we add the primitive functions we have defined above (line drawing, turtle movement, etc) to the list of basic Epic definitions, and convert the user defined procedures to Epic.

```
prims = basic_defs ++ [EpicFn (name "initSDL") initSDL,
                       EpicFn (name "drawLine") drawLine,
                       EpicFn (name "forward") forward, ... ]
```

We define a type class to capture conversion of expressions, commands and full programs into Epic terms. Programs maintain the turtle's state (an Epic `Expr`), and return a new state, so we pass this state to the compiler.

```
class Compile a where
    compile :: Expr -> a -> Term
```

In general, since we have set up all of the primitive operations as Epic functions, compiling an Atuin program consists of directly translating the abstract syntax to the Epic equivalent, making sure the state is maintained. For example, to compile a call we build an Epic function call and add the current state as the first argument. Epic takes strings as identifiers, so we use `fullId :: Id -> String` to convert an Atuin identifier to an Epic identifier.

```
compile state (Call i es) = app (fn (fullId i) @@ state) es
   where app f [] = f
         app f (e:es) = app (f @@ compile state e) es
```

Where operations are sequenced, we make sure that the state returned by the first operation is passed to the next:

```
compile state (Seq x y)
   = let_ (compile state x) (\state' -> compile state' y)
```

Atuin has higher order procedures which accept code blocks as arguments. To compile a code block, we build a function which takes the turtle state (that is, the state at the time the block is executed, not the state at the time the block is created). Epic's `effect_` function ensures that a closure is evaluated, but the result is not updated. Evaluating the closure may have side effects which may need to be executed again — consider the `repeat` function above, for example, where the code block should be evaluated on each iteration.

```
compile state (Block t) = term (\st -> compile st t)
compile state (Eval e)  = effect_ (compile state e @@ state)
```

The rest of the operations are compiled by a direct mapping to the primitives defined earlier. Finally, the main program sets up an SDL surface, creates an initial turtle state, and passes that state to the user-defined `main` function:

```
init_turtle surf = tuple_ @@ surf @@ int 320 @@ int 240 @@
                                     int 180 @@ col_white @@ bool True

runMain :: Term
runMain = let_ (fn "initSDL" @@ int 640 @@ int 480)
          (\surface ->
            (fn (fullId (mkId "main")) @@ (init_turtle surface)) +>
             flipBuffers surface +> pressAnyKey)
```

The full source code for Atuin and its compiler is available from Hackage (`http://hackage.haskell.org/package/atuin`), which covers the remaining technical details of linking compiled programs with SDL.

5 Related Work

Epic is currently used by Agda and Idris [5], as well as the development version of Epigram [7]. Initial benchmarking [6] shows that the code generated by Epic can be competitive with Java and is not significantly worse than C. Epic uses techniques from other functional language back ends [12, 15, 18] but deliberately exposes its core language as an API to make it as reusable as possible. Although there is always likely to be a trade off between reusability and efficiency, exposing the API will make it easier for other language researchers to build a new compiler quickly. As far as we are aware, Epic occupies a unique point in the design space of code generation tools — it is sufficiently high level that it captures common functional language abstractions without being so high level that it imposes constraints such as a type system on the language it is compiling. Alonzo, for example, is a prototype compiler for Agda [2] which compiles via GHC, but requires coercions in the generated code in order for it to be accepted by GHC's type checker. Coq's program extraction tool [10] also aims to generate executable code via a high level language, similarly requiring coercions. In contrast, systems such as the Lazy Virtual Machine [9], C-- [16] and LLVM [8] are designed as lower level target languages rather than high level APIs. We could nevertheless consider using these tools for Epic code generation.

6 Conclusion

Epic provides a simple path for language researchers to convert experimental languages (e.g. experimenting with new type systems or domain specific language design) into larger scale, usable tools, by providing an API for generating a compiler, dealing with well-understood but difficult to implement problems such as naming and scope management, code generation, interfacing with foreign functions and garbage collection. In this paper we have seen two examples of languages which can be compiled via Epic, both functionally based, but with different features. The high-level recipe for each is the same: define primitive functions as run-time support, then translate the abstract syntax into concrete Epic functions, using a combinator style API. In addition, we have implemented a compiler for λ_{Π} [11], a dependently typed language, which shows how Epic can handle languages with more expressive type systems[4].

Future work Since Epic is currently used in practice by a number of dependently typed functional languages, future work will have an emphasis on providing an efficient executable environment for these and related languages. An interesting research question, for example, is whether the rich type systems of these languages can be used to guide optimisation, and if so how to present the information gained by the type system to the compiler.

Currently, Epic compiles to machine code via C, using the Boehm conservative garbage collector [3]. While this has been reasonably efficient in practice, we believe that an LLVM based implementation [8, 19] with accurate garbage collection would be more appropriate as it could take advantage of functional language features such as immutability of data.

Perhaps more importantly, as a very simple functional language Epic is a convenient platform with which to experiment with functional compilation techniques. For example, we are developing an evaluator which will be a starting point for experimenting with supercompilation [13] and partial evaluation. Of course, any language which uses Epic as a back end will stand to gain from future optimisation efforts!

Acknowledgments

This work was partly funded by the Scottish Informatics and Computer Science Alliance (SICSA) and by EU Framework 7 Project No. 248828 (ADVANCE). Thanks to the anonymous reviewers for their constructive suggestions.

References

1. L. Augustsson. Cayenne - a language with dependent types. In *Proc. 1998 International Conf. on Functional Programming (ICFP '98)*, pages 239–250, 1998.

[4] `http://www.idris-lang.org/examples/LambdaPi.hs`

2. M. Benke. Alonzo — a compiler for Agda, 2007. Talk at Agda Implementors Meeting 6.
3. H.-J. Boehm, A. J. Demers, Xerox Corporation Silicon Graphic, and Hewlett-Packard Company. A garbage collector for C and C++, 2001.
4. E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
5. E. Brady. Idris — Systems programming meets full dependent types. In *PLPV*, pages 43–54, 2011.
6. E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 297–308, New York, NY, USA, 2010. ACM.
7. J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 3–14, New York, NY, USA, 2010. ACM.
8. C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, December 2002.
9. D. Leijen. LVM, the Lazy Virtual Machine. Technical Report UU-CS-2004-05, Institute of Information and Computing Sciences, Utrecht University, August 2005.
10. P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for proofs and programs*, LNCS. Springer, 2002.
11. A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.
12. 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.
13. N. Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 309–320, New York, NY, USA, 2010. ACM.
14. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, September 2007.
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, T. Nordin, and D. Oliva. C–: A portable assembly language. In C. Clack, editor, *Workshop on Implementing Functional Languages, St Andrews*. Springer-Verlag, 1997.
17. F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.
18. S. Smetsers, E. Nöcker, J. van Groningen, and R. Plasmeijer. Generating efficient code for lazy functional languages. In J. Hughes, editor, *Functional programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 592–617. Springer-Verlag, 1991.
19. 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.
20. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programm (ESOP 2009)*, pages 1–16, 2009.