Epic — a Generic Intermediate Functional Programming Language

Edwin C. Brady

School of Computer Science, University of St Andrews, St Andrews, Scotland.
Email: 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 fit the new language around another existing compiler.

Epic is a compiled functional language which exposes functional compilation techniques to a language implementor. It has both a concrete syntax and a Haskell API. It is independent of a source language's type system and semantics, supports eager or lazy evaluation, and has a range of primitive types and a lightweight foreign function interface. In this paper we describe Epic and demonstrate its flexibility by applying it to two very different functional languages: a dynamically typed turtle graphics language and a dependently typed lambda calculus.

1. Introduction

[Just some notes for now...]

Lots of backends for functional languages, e.g. STG [10, 13, 16], ABC [15]. But they aren't simple enough that they are easy to bolt on to a new language. Either too low level, or an interface isn't exposed, or where an interface is exposed, there are constraints on the type system. So things like Agda [12] have resorted to generating Haskell with unsafeCoerce, Cayenne [1] used LML with the type checker switched off. This works but we can't expect GHC optimisations without working very hard, are limited to GHC's choice of evaluation order, and could throw away useful information gained from the type system.

Epic originally written for Epigram [6] (the name[1] is short for "**Epi**gram **C**ompiler"). Now used by Idris [4], also as an experimental back end for Agda. It is specifically designed for reuse by other languages (in constrast to, say, GHC Core).

1.1 Features and non-features

Epic will handle the following:

[1] Coined by James McKinna

- Managing closures and thunks
- Lambda lifting
- Some optimisations (currently inlining, a supercompiler is planned)
- Marshaling values to and from foreign functions
- Garbage collection
- Name choices (optionally)

Epic will not do the following, by design:

- Type checking (no assumptions are made about the type system of the high level language being compiled)

Epic has few high level language features, but some additions will be considered which will not compromise the simplicity of the core language. For example, a pattern matching compiler is planned, and primitives for parallel execution.

Also lacking, but entirely possible to add later (with some care) are unboxed types.

1.2 Why an Intermediate Language

Why not generate Haskell, OCaml, Scheme, ...? In general they are too high level and impose design choices and prevent certain implementation choices. An intermediate level language such as Epic allows the following:

Control of generated code A higher level target language imposes implementation choices such as evaluation strategy and purity. Also makes it harder to use lower level features where it might be appropriate (e.g. while loops, mutation).

Control of language design Choice of a high level target language (especially a typed one) might influence our type system design, restrict our choices for ease of code generation.

Efficiency We might expect using a mature target language to give us optimisations for free. This might be true in many cases, but only if our source language is similar enough. e.g. in Epigram the type system tells us more about the code than we can pass on to a Haskell back end.

Epic 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, with the obvious exception that a functional style is encouraged! A further advantage of Epic is that the library provides *compiler combinators*, which guarantee that any output code will be syntactically correct and well-scoped.

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. As a shorthand, we use de Bruijn telescope notation, $\vec{x}$, to denote a sequence of x.

$$\begin{array}{rcll}
p & ::= & \vec{def} & \text{(Epic program)} \\
def & ::= & x(\vec{x}) = t & \text{(Top level definition)} \\
\\
t & ::= & x & \text{(Variable)} \\
 & \mid & t(\vec{t}) & \text{(Function application)} \\
 & \mid & \lambda x.\ t & \text{(Lambda binding)} \\
 & \mid & \underline{\text{let}}\ x\ =\ t\ \underline{\text{in}}\ t & \text{(Let binding)} \\
 & \mid & \mathsf{Con}\ i(\vec{t}) & \text{(Constructor application)} \\
 & \mid & t!i & \text{(Argument projection)} \\
 & \mid & t\ op\ t & \text{(Infix operator)} \\
 & \mid & \underline{\text{if}}\ t\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t & \text{(Conditional)} \\
 & \mid & \underline{\text{case}}\ t\ \underline{\text{of}}\ \vec{alt} & \text{(Case expressions)} \\
 & \mid & \underline{\text{lazy}}(t) & \text{(Lazy evaluation)} \\
 & \mid & \underline{\text{effect}}(t) & \text{(Evaluate an effectful term)} \\
 & \mid & \underline{\text{while}}\ t\ t & \text{(While loops)} \\
 & \mid & x := \ t\ \underline{\text{in}}\ t & \text{(Variable update)} \\
 & \mid & \underline{\text{foreign}}\ T\ str\ (\vec{t\ :\ T}) & \text{(Foreign call)} \\
 & \mid & \underline{\text{malloc}}\ t\ t & \text{(Manual allocation)} \\
 & \mid & i \mid f \mid c \mid b \mid str & \text{(Constants)} \\
\\
alt & ::= & \mathsf{Con}\ i(\vec{x})\ \mapsto\ t & \text{(Constructors)} \\
 & \mid & i\ \mapsto\ t & \text{(Integer constants)} \\
 & \mid & \underline{\text{default}}\ \mapsto\ t & \text{(Match anything)} \\
\\
op & ::= & + \mid - \mid \times \mid / \mid == \mid < \mid \leq \mid > \mid \geq \mid << \mid >> & \\
\\
x & ::= & \text{Variable name} & \\
i & ::= & \text{Integer literal} & \\
f & ::= & \text{Floating point literal} & \\
c & ::= & \text{Character literal} & \\
b & ::= & \text{Boolean literal } \mathsf{True} \mid \mathsf{False} & \\
str & ::= & \text{String literal} & \\
\\
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}$$

Figure 1. Epic syntax

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:

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

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*, i.e. applied to fewer arguments than they require. Evaluating a partially applied function results in a function which expects the remaining arguments.

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 $\vec{a}$ are the *fields*. The name Con is to suggest "Constructor". For example, we could represent a list using tagged unions:

- $\mathsf{Con}\ 0()$ representing the empty list.
- $\mathsf{Con}\ 1(x,\ xs)$ 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. For example, 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 $\underline{\text{lazy}}$ construct. An expression $\underline{\text{lazy}}(t)$ will not be evaluated until it is required by one of:

- Inspection in a $\underline{\text{case}}$ expression or the condition in an $\underline{\text{if}}$ statement.
- Field projection.
- Being passed to a foreign function.
- Explicit evaluation with $\underline{\text{effect}}$. This evaluates side-effecting code (and does not update the thunk).

Using $\underline{\text{lazy}}$ we can build an infinite list of values, ensuring that a value will not be added to the list until it is needed:

$$countFrom(x)\ =\ \mathsf{Cons}\ 1(x,\ \underline{\text{lazy}}(countFrom(x+1)))$$

We can safely write a function which takes the first n values from such a list, since the tail of the list will not be evaluated until it is needed:

$$\begin{array}{lll}
take(n, xs)\ =\ \underline{\text{case}}\ xs\ \underline{\text{of}} & & \\
\quad \mathsf{Con}\ 0() & \mapsto & \mathsf{Con}\ 0() \\
\quad \mathsf{Con}\ 1(x, xs') & \mapsto & \underline{\text{if}}\ n == 0\ \underline{\text{then}}\ \mathsf{Con}\ 0() \\
 & & \quad \underline{\text{else}}\ \mathsf{Con}\ 1(x, take(n-1, xs'))
\end{array}$$

$$vals()\ =\ take(10, countFrom(1))$$

Higher order functions

Finally, expressions may contain λ and $\underline{\text{let}}$ bindings. Higher order functions such as map are also permitted:

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

$$\begin{array}{l}
evens(n)\ =\ \underline{\text{let}}\ nums\ =\ take(n, countFrom(1))\ \underline{\text{in}} \\
\qquad map(\lambda x.\ x\ \times\ 2, nums)
\end{array}$$

Epic type	C Type
Int	`int`
Char	`char`
Bool	`int`
Float	`double`
String	`char*`
Ptr	`void*`
Unit	`void`
Any	`Closure*`

Table 1. Epic to C type conversion

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. Each primitive type corresponds to a C type, as shown in Table 1. Most of the conversions are straightforward mappings between the languages. Otherwise, Bool is represented by an C `int`, and Unit is used for the return type of `void` functions. Ptr is the type of pointer values originating in C. Values can also be passed in their Epic representation (`Closure*`) but the details are beyond the scope of this paper.

Epic values are represented differently from C values, so argument values must be converted before making a foreign call, and the return value converted back. Since Epic does no type checking or inference, a foreign call requires the argument and return types to be given explicitly. For example, consider the C sine function:

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

We can call this function from Epic by giving the C name, the return type (an Epic Float) and the argument type (also an Epic Float).

$$sin(x) \ = \ \underline{\mathsf{foreign}}\ \mathsf{Float}\ "\mathtt{sin}"\ (x\ :\ \mathsf{Float})$$

Often, dealing with foreign functions requires handling data types which originate in C. For example, opening a file uses the `fopen` function, which returns a pointer to the C type `FILE`:

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

We can use this function directly from Epic, using the Ptr type:

$$fopen(f, m) \ = \ \underline{\mathsf{foreign}}\ \mathsf{Ptr}\ "\mathtt{fopen}"(f : \mathsf{String}, m : \mathsf{String})$$

The String annotations in the foreign call ensure that the Epic string representation will be passed to C as a `char*`. Other file operations, such as writing or closing the file can refer directly to the file handle returned as a Ptr:

$$\begin{array}{lcl} fputs(x, h) & = & \underline{\mathsf{foreign}}\ \mathsf{Unit}\ "\mathtt{fputs}"(x : \mathsf{String}, h : \mathsf{Ptr}) \\ fclose(h) & = & \underline{\mathsf{foreign}}\ \mathsf{Unit}\ "\mathtt{fclose}"(h : \mathsf{Ptr}) \end{array}$$

Many foreign functions have side effects (in particular, I/O). Epic assumes that the high level language will arrange for side-effecting functions to be evaluated in the correct order. For this reason, it is important to understand the default evaluation strategy.

A note on memory management

When working with foreign functions it is important to consider how external code interacts with the garbage collector. In particular, *copying* garbage collectors can be problematic because there is a danger that a value referenced by external code may be moved, and there is no way for the collector to identify and move every reference in this code. This could cause a problem if the external library uses callbacks to Epic functions.

The current implementation uses a non-moving collector, the Boehm conservative collector for C [2], thus avoiding any problems. However, this is not a long term solution, as this kind of collector does not generally perform well for programs which create a lot of short lived objects (of which most functional programs are an example). Any future garbage collector for Epic will need to ensure that values originating in Epic are not copied once passed to foreign code.

2.3 Low Level Features

Epic emphasises control over safety, and therefore provides some low level features to give language implementations more control over generated code. 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.

Imperative features

Epic allows sequencing, while loops and variable update. Since the default evaluation strategy is strict, sequencing can be achieved simply with let bindings, discarding the variable:

$$\begin{array}{lcl} main \ = & \underline{\mathsf{let}}\ \_ & = \ putStr(\texttt{"Enter your name: "})\ \underline{\mathsf{in}} \\ & \underline{\mathsf{let}}\ n & = \ getStr()\ \underline{\mathsf{in}} \\ & & \ \ \ putStrLn(append(\texttt{"Hello "}, n)) \end{array}$$

We can also use while loops, and update variables. Variable update ($x \ := \ t_1\ \underline{\mathsf{in}}\ t_2$) behaves like a let binding, except that x must already be in scope, and the previous value of x is updated to be t_2. We can use these to write an imperative program such as the following, which prints numbers from 1 to 10:

$$\begin{array}{l} main \ = \ \underline{\mathsf{let}}\ c \ = \ 0\ \underline{\mathsf{in}} \\ \qquad\qquad \underline{\mathsf{while}}\ (c < 10) \\ \qquad\qquad\quad (c := c + 1\ \underline{\mathsf{in}} \\ \qquad\qquad\quad\ putStrLn(intToSring(c))) \end{array}$$

Memory allocation

The malloc construct allows manual 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. When evaluation is complete, the pool is freed, and the result copied. This can be valuable where an upper bound on memory usage can be predicted accurately (whether manually or automatically) as it reduces the garbage collection overhead.

2.4 Haskell API

The primary interface to Epic is through a Haskell API, which is used to build expressions and programs, and to compile them to executables. Implementing a compiler for a high level language is then a matter of 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

Build expressions, with a name supply. `Expr` is the internal (abstract) representation of Epic expressions:

```
type Term = State Int Expr

class EpicExpr e where
    term :: e -> Term
```

Instances for raw expressions and terms. More interestingly, an instance for functions which allows Haskell functions to be used to build Epic functions without worring about scope. `R` and `Lam` are internal representations for references and λ bindings respectively.

```
instance (EpicExpr e) => EpicExpr (Expr -> e) where
    term f = do var <- get
                put (var+1)
                let arg = MN "evar" var
                e' <- term (f (R arg))
                return (Lam arg e')
```

Aside: Internally, names are separated into user supplied names, and names invented by the machine. Using machine names guarantees we won't clash with a user name, and we can give an annotation (`"evar"` here) which says where the name arose.

```
data Name = UN String     -- user name
          | MN String Int -- machine generated name
```

It can be more convenient to give explicit names for λ bindings, so we declare an instance to allow that:

```
instance EpicExpr e => EpicExpr ([Name], e) where
    term (ns, e) = do
        e' <- term e
        foldM (\e n -> return (Lam n e)) e' ns
```

Both forms can be mixed freely. A program is a collection of named Epic declarations:

```
data EpicDecl = forall e. EpicExpr e => EpicFn Name e
              | ...

type Program = [EpicDecl]
```

Epic declarations are usually just a function (`EpicFn`) but can also be used to include C header files, declare libraries for linking and declare types for exporting as C types. The library also provides a number of built in definitions for some common operations such as outputting strings and converting data types:

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

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've seen λ bindings, using either Haskell's λ or pairing the names with their scope.

General form is that we build a `Term` (i.e. an expression managing the name supply) by combining arbitrary Epic expressions (i.e. instances of `EpicExpr`).

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

Distinction between Haskell application and Epic application (`@@`).

Operators. Separate versions for floating point and integer.

```
plus_, minus_, times_, divide_,    :: Op
plusF_, minusF_, timesF_, divideF_ :: Op
lt_, lte_, gt_, gte_,              :: Op
ltF_, lteF_, gtF_, gteF_,          :: Op
shiftl_, shiftr_                  :: Op
```

Convention: Epic keywords are represented by a Haskell function which is the keyword with an underscore suffix. Arises because we can't have "let", "if", "case" etc as function names, and extended to other primitives such as operators, foreign calls (anything that'd be a keyword in general).

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

For let bindings, we can either use higher order syntax or bind an explicit name:

```
let_  :: (EpicExpr e) =>
         e -> (Expr -> Term) -> Term
letN_ :: (EpicExpr val, EpicExpr scope) =>
         Name -> val -> scope -> Term
```

Shorthand if we're just using lets to sequence then throw away the bound thing:

```
(+>) :: (EpicExpr c) => c -> Term -> Term
(+>) c k = let_ c (\x -> k)
```

Name management:

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

Building constructor forms. `tuple_` is provided as a shorthand if the tag is not important.

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

Case analysis

Case expressions:

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

Building case alternatives for constructors. We use the same trick as we did for λ-bindings, either allowing Haskell to manage to scope of constructor arguments, or giving names explicitly, or a mixture.

```
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, tuples, or integer constants, as well as a default case if all other alternatives fail to match. In each of the following, `e` is the right hand side. For constructors and tuples, arguments may be bound in the match. (We have to use `Alternative` rather than `EpicExpr` to ensure that we get arguments bound in the match, rather than a λ binding on the right hand side).

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

A complete example

We have enough to write a simple example now. Here is the *map* function from earlier. Note the distinction between `con_`, used to build a constructor, and `con`, used to introduce a match alternative.

```
map_ :: Expr -> Expr -> Term
map_ f xs = case_ xs
   [con 0 (con_ 0),
    con 1 (\ (x :: Expr) (xs' :: Expr)
      -> con_ 1 @@ (f @@ x) @@ (fn "map" @@ f @@ xs'))]
```

In the recursive call, we refer to the function by an Epic name, rather than a Haskell name. So we need to ensure that these names are consistent. We can declare that the name "map" refers to the definition map_:

```
mapDef :: EpicDecl
mapDef = EpicFn (name "map") map_
```

We have a function to add up all the integers in a list:

```
sum_ :: Expr -> Term
sum_ xs = case_ xs
            [con 0 (int 0),
             con 1 (\ (x :: Expr) (xs' :: Expr) ->
                        op_ plus_ x (fn "sum" @@ xs'))]
```

This constructs the Epic function:

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

We write a test funtion which doubles all the elements in a list, then computes the sum of the result. Note that we have written the function argument to map as an inline function. The main program simply outputs the result of this function.

```
test_ = fn "sum" @@
       (fn "map" @@ \ (x :: Expr) -> op_ times_ x (int 2)
              @@ (con_ 1 @@ (int 5) @@
                    (con_ 1 @@ (int 10) @@ con_ 0)))

main_ = fn "putStrLn" @@ (fn "intToString" @@ fn "test")
```

test_ constructs the Epic function:

$$\begin{array}{lll} test & = & sum(map(\lambda x.x\ \times\ 2, \\ & & \quad \mathsf{Con}\ 1(5, \mathsf{Con}\ 1(10, \mathsf{Con}\ 0())))) \end{array}$$

Finally, we give a complete list of definitions, mapping concrete names to the functions we've written.

```
defs = basic_defs ++ [EpicFn (name "main") main_,
                      EpicFn (name "map") map_,
                      EpicFn (name "sum") sum_,
                      EpicFn (name "test") test_]
```

To compile and run this, we use the run function from the Epic API, which builds an executable and runs it. The following program outputs 30.

```
main = run defs
```

3. Example High Level Languages

In this section we present compilers for three different high level languages to demonstrate aspects of the Epic API. Firstly, we present a compiler for the untyped λ-calculus using Higher Order Abstract Syntax, which shows the fundamental features of Epic required to implement a complete compiler. Secondly, we present a compiler for λ_Π [9], a dependently typed language, which shows how Epic can handle languages with more expressive type systems. Finally, we present a compiler for a dynamically typed graphics language, which shows how Epic can be used for languages with run-time type checking and which require foreign function calls.

3.1 Untyped λ-calculus

3.1.1 Representation

Our first example is an implementation of the untyped λ-calculus, plus primitive integers and strings, and arithmetic and string operators. The language is represented in Haskell using higher order abstract syntax (HOAS). That is, we represent λ-bindings (Lam) as Haskell functions, using a Haskell variable name to refer to the locally bound variable. 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)]
```

3.1.2 Compilation

Our aim is to convert a collection of Defs into an executable, using one of the following functions from the Epic API:

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

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, then, 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.

The cases worth noting are the compilation of λ-bindings and string concatenation. 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

```
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_
          eOp Times  = times_
          eOp Divide = divide_
          eOp Eq     = eq_
          eOp Lt     = lt_
          eOp Gt     = gt_
```

Figure 2. Compiling Untyped λ-calculus

the equivalent reference to the name in Epic. While there may be neater solutions involving an environment, or avoiding HOAS, this solution is very 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 function called `"main"` or Epic will report an error.

```
mkProgram :: Defs -> Program
mkProgram ds = basic_defs ++
               map (\ (n, d) -> EpicFn n (build d)) ds

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

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

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

This is a compiler for a very simple language, but a compiler for any more complex language using the Epic API 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`.

3.2 Dependently Typed λ-calculus

λ_Π [9]. Complications: elimination operators. Representation uses de Bruijn indices. Need a way to dump output. Non-complication: odd type system.

3.2.1 Representation

3.2.2 Compilation

3.3 A Dynamically Typed Turtle Graphics Language

4. Implementation

How it's implemented is not really important to the user — a compiler can target Epic without knowing, and we could drop in a new back end at any time in principle.

There is currently one back end, but more are planned. Compiled via C. Garbage collection with Boehm [2], `%memory`. (Note that a non-moving collector makes things easier for foreign functions, but may not be the best choice in the long run).

Later plans: compile via LLVM, allow plug in garbage collectors (important for embedded systems, device drivers, operating system services, for example).

5. Performance

6. Related Work

Should mention Agda and Idris [5] and the development version of Epigram [6] as having Epic back ends.

GHC's run-time system [10, 13], ABC machine [15] and why we don't just use one of them (no useful interface, imposes constraints on the type system). Some influence from GRIN [3].

Lazy virtual machine [8]. C- [14] and LLVM [7] as possible code generation strategies. Supercompilation for optimisations [11].

7. Conclusion

References

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

[2] H.-J. Boehm, A. J. Demers, Xerox Corporation Silicon Graphic, and Hewlett-Packard Company. A garbage collector for C and C++, 2001.

[3] U. Boquist and T. Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, pages 58–84, 1996.

[4] E. Brady. Idris — systems programming meets dependent types. In *Programming Languages Meets Program Verification*, 2011.

[5] 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.

[6] 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.

[7] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, December 2002.

[8] D. Leijen. Lvm, the lazy virtual machine, November 2002.

[9] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.

[10] 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.

[11] 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.

[12] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, September 2007.

[13] 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.

[14] 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.

[15] 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.

[16] 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.