Epic — a Generic Functional Compiler

Edwin Brady

December 14, 2010

Abstract

Epic is a minimal, compiled functional language, intended as a flexible target language for high level functional languages. It is independent of the source language's type system, supports eager or lazy evaluation, and has a range of primitive types and a lightweight foreign function interface.

1 Introduction

Lots of backends for functional languages, e.g. STG [4, 6, 8], ABC [7]. 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 [5] have resorted to generating Haskell with unsafeCoerce.

Epic originally written for Epigram [3] (the name is short for "**Epi**gram **C**ompiler"). Now used by Idris [2], also as an experimental back end for Agda.

2 The Epic Language

2.1 Definitions

BNF in Figure 1. Expressions. `let`, `case`, `lazy`, `while`

2.2 Types

`Int`, `Float`, `Bool`, `Data`, `Ptr`, `Unit`. Unchecked! Used for marshalling foreign functions only.

Run-time representation (31 bit ints).

2.3 Foreign Functions

Calling, exporting.

2.4 Implementation

How it's implemented is not really important — a compiler can target Epic without knowing. There is currently one back end, but more are planned. Compiled via C. Garbage collection with Boehm [1], `%memory`.

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

2.5 Haskell API

(Give HOAS translation as an example)

3 Example High Level Languages

[Give high level translation, rather than concrete Haskell]

4 Performance

5 Related Work

6 Conclusion

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] D. A. Terei and M. M. Chakravarty. An LLVM backend for ghc. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.

$$
\begin{array}{rcll}
t & ::= & x & \text{(Variable)} \\
& | & t(\vec{t}) & \text{(Function application)} \\
& | & \lambda x : T.\ t & \text{(Lambda binding)} \\
& | & \underline{\text{let}}\ x\ :\ T\ =\ t\ \underline{\text{in}}\ t & \text{(Let binding)} \\
& | & \mathsf{Con}\ i(\vec{t}) & \text{(Constructor application)} \\
& | & t!i & \text{(Argument projection)} \\
& | & t\ op\ t & \text{(Infix operator)} \\
& | & \underline{\text{if}}\ t\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t & \text{(Conditional)} \\
& | & \underline{\text{while}}(t, t) & \text{(While loops)} \\
& | & \underline{\text{case}}\ t\ \underline{\text{of}}\ \vec{alt} & \text{(Case expressions)} \\
& | & \underline{\text{lazy}}(t) & \text{(Lazy evaluation)} \\
& | & \underline{\text{foreign}}\ T\ str\ (t\ \vec{:}\ T) & \text{(Foreign call)} \\
& | & i \mid f \mid c \mid b \mid str & \text{(Constants)}
\end{array}
$$

$$
\begin{array}{rcll}
alt & ::= & \mathsf{Con}\ i(\vec{x})\ \mapsto\ t & \text{(Constructors)} \\
& | & i\ \mapsto\ t & \text{(Integer constants)} \\
& | & \underline{\text{default}}\ \mapsto\ t & \text{(Match anything)}
\end{array}
$$

$$
op\ ::=\ + \mid - \mid \times \mid / \mid == \mid < \mid \leq \mid > \mid \geq
$$

$$
\begin{array}{rcll}
T & ::= & \mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Bool} \mid \mathsf{Float} \mid \mathsf{String} & \text{(Primitives)} \\
& | & \mathsf{Unit} & \text{(Unit type)} \\
& | & \mathsf{Ptr} & \text{(Foreign pointers)} \\
& | & \mathsf{Fun} & \text{(Any function type)} \\
& | & \mathsf{Data} & \text{(Any data type)} \\
& | & \mathsf{Any} & \text{(Unchecked polymorphic type)}
\end{array}
$$

$$
\begin{array}{rcl}
x & ::= & \text{Variable name} \\
i & ::= & \text{Integer literal} \\
f & ::= & \text{Floating point literal} \\
c & ::= & \text{Character literal} \\
b & ::= & \text{Boolean literal } \mathsf{True} \mid \mathsf{False} \\
str & ::= & \text{String literal}
\end{array}
$$

$$
def\ ::=\ x(x\ \vec{:}\ T) \rightarrow T = t \quad \text{(Top level definition)}
$$

Figure 1: Epic syntax