Epigram (programming language)

Epigram is a functional programming language with dependent types, and the integrated development environment (IDE) usually packaged with the language. Epigram's type system is strong enough to express program specifications. The goal is to support a smooth transition from ordinary programming to integrated programs and proofs whose correctness can be checked and certified by the compiler. Epigram exploits the *Curry–Howard correspondence*, also termed the *propositions as types principle*, and is based on intuitionistic type theory.

The Epigram prototype was implemented by Conor McBride based on joint work with James McKinna. Its development is continued by the Epigram group in Nottingham, Durham, St Andrews, and Royal Holloway, University of London in the United Kingdom (UK). The current experimental implementation of the Epigram system is freely available together with a user manual, a tutorial and some background material. The system has been used under Linux, Windows, and macOS.

It is currently unmaintained, and version 2, which was intended to implement Observational Type Theory, was never officially released but exists in GitHub.

Epigram

Paradigm	Functional
Designed by	Conor McBride James McKinna
Developer	*Unmaintained*
First appeared	2004
Stable release	1 / October 11, 2006
Typing discipline	strong, static, dependent
OS	Cross-platform: Linux, Windows, macOS
License	MIT[1]
Website	web.archive.org/web/20120717070845/http://www.e-pig.org/darcs/Pig09/web/ (https://web.archive.org/web/20120717070845/http://www.e-pig.org/darcs/Pig09/web/)
Influenced by	
ALF	
Influenced	
Agda, Idris	

Contents

Syntax

Epigram uses a two-dimensional, natural deduction style syntax, with versions in LaTeX and ASCII. Here are some examples from *The Epigram Tutorial*:

Examples

The natural numbers

The following declaration defines the natural numbers:

```
     (         !       (          !   (  n : Nat  !
data !---------! where !----------! ; !-----------!
     ! Nat : * )       !zero : Nat)   !suc n : Nat)
```

The declaration says that `Nat` is a type with kind `*` (i.e., it is a simple type) and two constructors: `zero` and `suc`. The constructor `suc` takes a single `Nat` argument and returns a `Nat`. This is equivalent to the Haskell declaration "`data Nat = Zero | Suc Nat`".

In LaTeX, the code is displayed as:

$$\underline{\textbf{data}}\ \left(\frac{}{\textsf{Nat} : \star}\right)\ \underline{\textbf{where}}\ \left(\frac{}{\textsf{zero} : \textsf{Nat}}\right)\ ;\ \left(\frac{n : \textsf{Nat}}{\textsf{suc}\ n : \textsf{Nat}}\right)$$

The horizontal-line notation can be read as "assuming (what is on the top) is true, we can infer that (what is on the bottom) is true." For example, "assuming `n` is of type `Nat`, then `suc n` is of type `Nat`." If nothing is on the top, then the bottom statement is always true: "`zero` is of type `Nat` (in all cases)."

Recursion on naturals

$$\begin{aligned}&\forall P : \textsf{Nat} \to \star \Rightarrow P\ \textsf{zero} \to \\ \textsf{NatInd} :\ &(\forall n : \textsf{Nat} \Rightarrow P\ n \to P\ (\textsf{suc}\ n)) \to \\ &\forall n : \textsf{Nat} \Rightarrow P\ n\end{aligned}$$

$$\textsf{NatInd}\ P\ mz\ ms\ \textsf{zero} \equiv mz$$

$$\textsf{NatInd}\ P\ mz\ ms\ (\textsf{suc}\ n) \equiv ms\ n\ (NatInd\ P\ mz\ ms\ n)$$

...And in ASCII:

```
NatInd : all P : Nat -> * => P zero ->
         (all n : Nat => P n -> P (suc n)) ->
         all n : Nat => P n
NatInd P mz ms zero => mz
NatInd P mz ms (suc n) => ms n (NatInd P mz ms n)
```

Addition

$$\begin{aligned}&\textsf{plus}\ x\ y \Leftarrow \underline{\textsf{rec}}\ x\ \{ \\ &\quad \textsf{plus}\ x\ y \Leftarrow \underline{\textsf{case}}\ x\ \{ \\ &\quad\quad \textsf{plus zero}\ y \Rightarrow y \\ &\quad\quad\quad \textsf{plus}\ (\textsf{suc}\ x)\ y \Rightarrow \textsf{suc}(\textsf{plus}\ x\ y)\ \}\ \}\end{aligned}$$

...And in ASCII:

```
plus x y <= rec x {
  plus x y <= case x {
    plus zero y => y
    plus (suc x) y => suc (plus x y)
  }
}
```

Dependent types

Epigram is essentially a typed lambda calculus with generalized algebraic data type extensions, except for two extensions. First, types are first-class entities, of type $\star$; types are arbitrary expressions of type $\star$, and type equivalence is defined in terms of the types' normal forms. Second, it has a dependent function type; instead of $P \to Q$, $\forall x : P \Rightarrow Q$, where x is bound in Q to the value that the function's argument (of type P) eventually takes.

Full dependent types, as implemented in Epigram, are a powerful abstraction. (Unlike in Dependent ML, the value(s) depended upon may be of any valid type.) A sample of the new formal specification capabilities dependent types bring may be found in *The Epigram Tutorial*.

See also

- ALF, a proof assistant among the predecessors of Epigram.

Further reading

- McBride, Conor; McKinna, James (2004). "The view from the left". *Journal of Functional Programming*. **14**: 69–111. doi:10.1017/S0956796803004829 (https://doi.org/10.1017%2FS0956796803004829).
- McBride, Conor (2004). The Epigram Prototype, a nod and two winks (Report).
- McBride, Conor (2004). The Epigram Tutorial (Report).
- Altenkirch, Thorsten; McBride, Conor; McKinna, James (2005). Why Dependent Types Matter (Report).
- Chapman, James; Altenkirch, Thorsten; McBride, Conor (2006). Epigram Reloaded: A Standalone Typechecker for ETT (Report).
- Chapman, James; Dagand, Pierre-Évariste; McBride, Conor; Morris, Peter (2010). The gentle art of levitation (Report).

External links

- Official website (http://e-pig.org)
- Epigram 1 (https://github.com/david-christiansen/epigram1) on GitHub
- Epigram2 (https://github.com/mietek/epigram2) on GitHub
- EPSRC (https://web.archive.org/web/20060209235723/http://www.macs.hw.ac.uk/~fairouz/projects/EffProClaLog.html) on ALF, lego and related; archived version from 2006

References

1. "Epigram – Official website" (https://code.google.com/p/epigram/). Retrieved 28 November 2015.