Epigram

Eelis van der Weegen

2007-06-15

Epigram

Epigram is a combination of:

- A clean functional programming language;
  - with dependent types
  - Curry-Howard (propositions as types, programs as proofs, etc)
  - intuitionistic type theory
  - inductively defined data types (as in Coq)

- An integrated editor-typechecker-interpreter.
  $\rightarrow$ Built on top of xemacs.

Epigram

Epigram is a combination of:

- A clean functional programming language;
  - with dependent types
  - Curry-Howard (propositions as types, programs as proofs, etc)
  - intuitionistic type theory
  - inductively defined data types (as in Coq)

- An integrated editor-typechecker-interpreter.
  $\longrightarrow$ Built on top of xemacs.

Epigram

Main goal: To bring functional programming to a new level by introducing dependent types *and* "proper" support for programming with them.

Note: Not primarily a proof assistant, although programming with dependent types inevitably moves in that direction.

Versions / history

Epigram 1 (+/- 2004)

By Conor McBride (Univ. of Nottingham), James McKinna (Univ. of St Andrews).

Design influenced by:

- Lego (late ’90s), proof assistant implementing various related type systems (e.g. LF, CC, GCC, UTT).
- ALF (early ’90s), system combining functional and logic programming techniques. Has certain pattern-matching features.

Versions / history

Epigram 1 is very much an early prototype:

- incomplete
- unstable
- no standard library to speak of
- minimal error reporting
- only small Epigram programs/proofs developed so far

Development focus is on Epigram 2, which will be a complete reimplementation with various new features.

Dependent types - refresher

Idea: generalize ordinary function type $A \to B$ into:

$$\forall x : A \Rightarrow B.$$

Crucial: x may occur in B! (If it doesn't, $A \to B$ may be used as a shorthand for $\forall x : A \Rightarrow B$.)

Example (assume *Vec n* is the type of lists of *Nat*s, of length *n*):

$$repeat : Nat \to \forall n : Nat \Rightarrow Vec\ n$$

$$repeat\ \ 6\ \ 4 = [6, 6, 6, 6]$$

(Demo 1: *repeat*)

Dependent types - refresher

Idea: generalize ordinary function type $A \to B$ into:

$$\forall x : A \Rightarrow B.$$

Crucial: x may occur in B! (If it doesn't, $A \to B$ may be used as a shorthand for $\forall x : A \Rightarrow B$.)

Example (assume *Vec n* is the type of lists of *Nat*s, of length *n*):

$$repeat : Nat \to \forall n : Nat \Rightarrow Vec\ n$$

$$repeat\ \ 6\ \ 4 = [6, 6, 6, 6]$$

(Demo 1: *repeat*)

Dependent types - refresher

Idea: generalize ordinary function type $A \to B$ into:

$$\forall x : A \Rightarrow B.$$

Crucial: x may occur in B! (If it doesn't, $A \to B$ may be used as a shorthand for $\forall x : A \Rightarrow B$.)

Example (assume *Vec n* is the type of lists of *Nat*s, of length *n*):

$$repeat : Nat \to \forall n : Nat \Rightarrow Vec\ n$$

$$repeat\ \ 6\ \ 4 = [6, 6, 6, 6]$$

(Demo 1: *repeat*)

Consequences of dependent types (1)

Dependent types allow for radically more expressive types (and therefore propositions)!

But.. writing programs that make effective use of dependent types can be challenging. For example:

- Computation at the type level becomes commonplace:

$$concat : \forall m\ n : Nat, a : Vec\ m, b : Vec\ n \Rightarrow Vec\ (m + n)$$

(Demo 2: *concat*)

Consequences of dependent types (2)

- Type equality becomes an issue: $Vec\ (m + n)$ and $Vec\ (n + m)$ represent the same type, but this requires proof unless m and n are known.
- Case analysis can/must become more sophisticated.

Epigram aims to deal with these issues, to make programming with dependent types manageable.

(Demo 3: pattern matching)

Consequences of dependent types (2)

- Type equality becomes an issue: $Vec\ (m + n)$ and $Vec\ (n + m)$ represent the same type, but this requires proof unless m and n are known.
- Case analysis can/must become more sophisticated.

Epigram aims to deal with these issues, to make programming with dependent types manageable.

(Demo 3: pattern matching)