Expressive Programming Language Design: Organizational Meeting

Adrian King
2 November 2015

What Makes an Expressive Language?

A language is expressive when it lets you say what you mean and mean what you say.

Here are some ways you might group the components of what makes up “expressivity”:

- Syntactic and semantic structure
- Supporting infrastructure
- What the language can say about itself: self-reference

Syntactic and Semantic Structure

Syntactic and semantic structure are what many people think of as the “language proper”. Here are some characteristics that contribute to expressivity:

- Concision
- Familiarity
- Orthogonality

But you can also make arguments that these characteristics make a language *harder* to understand.

Infrastructure

The “language proper” is not the only determinant of how a language is perceived. Very important are:

- Libraries
- Tools
- Interoperability
- Community

Self-Reference

Languages differ in how easy it is for a program to talk about a program (maybe itself):

- Self-construction: how programs build programs (`eval`, `quote`, macros, etc.)
- Self-examination (reflection)
- Self-description (static types in general, dependent types in particular)

Also, parametricity in general lets programs manipulate programs, but what sorts of parameters are allowed?

Some Languages

Some languages we might investigate:

- Proofy: Coq, Agda, Idris, NuPRL, Twelf
- Logicky: Prolog, Twelf (?), Mercury, Oz
- Classics: APL, Smalltalk, Racket (Scheme)
- Functiony: Racket, Haskell, Scala
- Concurrenty: Go, Oz, Erlang, Elixir, etc.
- Buzzy: Rust
- Hybridy: Sage

What language(s) do you want to see here?

Some Tasks

- Define a data type (or data structure) that represents natural numbers in unary form. Implement addition, multiplication, and factorial, and show that 3! = 6. Define evenness and oddness, and prove that 6 is even.
- Define lists and implement concatenation.
- Given a binary operation that is commutative and associative, implement a parallel reduce for it on a nonempty collection.
- Write a typechecker and interpreter for the simply typed lambda calculus.

Some Concepts I'm Trying to Wrap My Head Around Right Now

How much stuff do you need to prove in a practical (not primarily for proving stuff) programing language? How do different approaches to these things affect what kind of mathematics you can do, and do you care?:

- Higher equalities (HoTT)
- Stratification of universes
- Inductive types plus (constrained) recursion vs. Martin-Löf types and induction principles
- W types: WTF? types
- Objects/modules: are they just functions?
- First-class refinements: just a pair, with one component inferrable?

An Idea for a Language Built in Layers

source (with infix operators)	⇒ s-expressions
s-expressions	⇒ s-expressions (macros)
s-expressions	⇒ typed intermediate language with named variables (more macros)
typed intermediate language with named variables	⇒ typed intermediate language with DeBruijn indices
typed intermediate language with DeBruijn indices	⇒ typed intermediate language with DeBruijn indices (typechecking and inference)
typed intermediate language with DeBruijn indices	⇒ untyped lambda calculus (augmented with type information as necessary for reflection/pattern matching)
untyped lambda calculus	⇒ interpreted, or compiled to machine code

Untyped Lambda Calculus Interpreter in Rofl, a Prolog-Like Language

```
;;
;; The interpreter proper:
;;

Var [] / (hd |: _) => hd
Var (1 |: n) / (_ |: tl) => val ~|
  Var n / tl => val

Lam body / stk => Clos stk body

App fun arg / stk => val ~|
  fun / stk => Clos cstk body,
  arg / stk => arg1,
  body / (arg1 |: cstk) => val

;;
;; Literals (C is for Constant):
;;

C value / stk => value
```

```
;;
;; S-expression manipulation:
;;

(hd :: tl) / stk => (hd1 |: tl1) ~|
  hd / stk => hd1,
  tl / stk => tl1

Mat matchee pattern ifMat ifNoMat / stk => val ~|
  matchee / stk => m,
  (m `Matches pattern / stk >> stk1)
  ?? (ifMat / stk1 => val)
  !! (ifNoMat / stk => val)

val `Matches (C val) / stk >> stk
val `Matches Var / stk >> (val |: stk)
(h |: t) `Matches (hpat :: tpat) / stk >> stk2 ~|
  h `Matches hpat / stk => stk1,
  t `Matches tpat / stk1 => stk2
```

Some Poorly-Thought-Out Syntax Examples for a Dependently Typed Functional Language with First-Class Refinements

```
fact ~
  0\ 1 |
  n\ n * fact n-1
;; Type: (n: Nat, n = 0)? (n: Nat, n = 1) |_| (n: Nat, n <> 0)? (r: Nat, r = n * fact n - 1)

zip ~
  Nil\ Nil\ Nil |
  (x :: xs)\ (y :: (ys :| len ys = len xs))\ x,y :: zip xs ys
;; Type:
    (a b: Ty)?*? {
      (nil: List a, nil = Nil)? (nil: List b, nil = Nil)? (nil: List a&b, nil = Nil) |_|
      (x::xs: List a)? (y::ys: List b, len ys = len xs)? (r: List a&b, r = x,y :: zip xs ys)
    }
```

Dependent types can be very informative (and very verbose).

Untagged Type Unions and Subtyping—A Natural Fit with Refinements?

```
SExp ~ Symbol + List SExp

;; Symbol <: SExp

;; List SExp <: Sexp


sexpSize ~
  Nil\ 0 |
  (_: Symbol)\ 1 |
  hd :: tl\ sexpSize hd + sexpSize tl
```

From *Design Principles Behind Smalltalk*, Daniel H. H. Ingalls, 1981...

- Personal Mastery: If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.
- Good Design: A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.
- Purpose of Language: To provide a framework for communication.
- Scope: The design of a language for using computers must deal with internal models, external media, and the interaction between these in both the human and the computer.

...continued...

- Objects: A computer language should support the concept of “object” and provide a uniform means for referring to the objects in its universe.
- Storage Management: To be truly “object-oriented”, a computer system must provide automatic storage management.
- Messages: Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.
- Uniform Metaphor: A language should be designed around a powerful metaphor that can be uniformly applied in all areas.
- Modularity: No component in a complex system should depend on the internal details of any other component.

...and continued...

- Classification: A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the kernel classes of the system.
- Polymorphism: A program should specify only the behavior of objects, not their representation.
- Factoring: Each independent component in a system would appear in only one place.
- Leverage: When a system is well factored, great leverage is available to users and implementers alike.
- Virtual Machine: A virtual machine specification establishes a framework for the application of technology.
- Reactive Principle: Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.

...and finally:

- Operating System: An operating system is a collection of things that don't fit into a language. There shouldn't be one.