The Constructive Foundations of the System *eb*

D. E. (Steve) Stevenson
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

steve@cs.clemson.edu
http://www.cs.clemson.edu/~steve/

I Want a Language...

...That has

- WYSIWYG semantics of Fortran.

- Has a proof theory similar to mathematical practice.

`eb` is a functional & logic language

- Using constructive and intuitionistic semantics for CSE.

- Finite Algorithms not countable.

- Finite number system.

What Work Am I Asking For?

1. Pre-theoretical understanding.
2. Language development.
3. Philosophical foundations.
4. Choice of foundations (interrelated and in parallel):
   (a) Choice of formal logical system. *E.g.*, Troelstra's system.
   (b) Choice of semantical system (for representation) and interpretation.
   (c) Choice of computational system.
   (d) Enunciation of proof theory.
5. Formalizing selected pre-theoretical ideas.
6. Fundamental Ontology: Objects, operations, relations. Sets and presets. Numbers. Functions and relations. Formal and informal constructive proofs.
7. Informal Axiomatics
8. Proof-theoretic enunciation of principles
9. Non-logical axioms and other logical foundations.
10. Non-logical definitions.
11. Computational Definitions.
12. Development of the theory.

Problem Areas

- What to do in a computer program when something is not computable? We have chosen to `fail` rather than to specify a user-implemented protocol.

- What about the mixed symbolic/number manipulation aspects? We have chosen to implement both aspects, in effect, mixing relational and functional capabilities.

- Different types should not be different programs, necessarily.

A Sidetrack

- Science is done in classical logic.

- Pre- and post- conditions are in classical logic BUT programs constructive.

- Need to develop an understanding of how to be constructive.

Classical logic came from Euclid. We presented new insights in Euclidean geometry with constructive foundations.

Where Bishop Fits In

1. Intensional School: Floyd-Hoare-Dijkstra.

2. Extensional School:

   (a) Recursive Analysis **RA**: λ-calculus and classical logic

   (b) Martin-Löf Theories **MT**: Type theory

   (c) Bishop-Bridges-Beeson **BBB**: Intuitionistic logic, constructive math

   All believe Church's Thesis.

Differences

Differences revolve around ontology

- Intensional school: Unicorns.
- Extensional school: Nothing exists unless constructed.

So What?

- *Constructive Conservation of Existence*: objects exist only when constructed and carry only structures which have been constructed.

- The programming equivalent to impredication is call "Segment Fault".

- *Not computable* is a legitimate answer.

- New concept: intensional items...items that are not yet verified as constructed. Example: roots of equations.

$$\{\propto x | Px\}.$$

Church's Thesis

- An interpretation: every object must have a concrete description or itself be concrete.

- Rules like $\forall x Px \to Pa$ must be accompanied by the substitution of a concrete object.

A *notion* η_A is a decidable property of predicate A. We know something only in the case we have a notion for it.

Sets

Constructing an object in **BBB**.

1. describe a process to construct an arbitrary object;

2. describe a process for testing equality; and

3. verify that the equality relation so defined is indeed an equivalence relation.

A (classical) set without equality is called a *preset*.

Constructive *versus* **BBB**

- A proof is any completely convincing argument.
- Proofs must have [computational] content.
- Concrete objects or constructions only (witness).
- *Set* is a preset equipped with *equality*.
- No excluded middle or impredications.
- No complete infinities.
- No assumptions of omniscience.

Propositions

Propositions are presets whose elements are proofs. A proposition of a statement A is constructed by describing what must be done to construct a proof of A in a completely convincing manner.

Logical Rules

Proofs work on objects. $p \vdash P$ if

$p \vdash A \wedge B$	p is a pair $< r, s >$ with $r \vdash A$ and $s \vdash B$
$p \vdash A \vee B$	p is a pair $< 0, q >$ with $q \vdash A$ or p is $< 1, q >$ and $q \vdash B$ Counterexample: Law of Addition
$p \vdash A \rightarrow B$	p is a *rule* q which transforms any proof s of A into a proof $q(s)$ of B plus any supplemental information needed.
$p \vdash \perp$	is impossible
$p \vdash \neg A$	Is the same as $p \vdash A \rightarrow \perp$
$p \vdash \exists x A x$	p is the pair $< x, q >$ for any $x \in X^{\ddagger}$ and $q \vdash Ax$. **Observe ontology restrictions**
$p \vdash \forall x A x$	p is a rule q such that for all $x \in X^{\ddagger}$ we have that $qx \vdash Ax$ plus supplemental information.

Mathematics is not a meaningless game

A proposition is *meaningful* in that they supply a completely genuine description. The constructivist view is that certain formal propositions in classical mathematics are not meaningful; hence, classical proof-theoretic derivability can be deficient in meaning.

The Feferman Criteria

M is a body of of informal mathematics.

> [A theory] T is an *adequate formalization* of M if every concept, argument and result of M may be represented by a (basic or defined) concept, proof, and theorem, respectively, of T.

> [A theory] T is *in accordance with* or *faithful to* a M if every basic concept of T corresponds to a basic concept of M and every axiom and rule of T corresponds to or is implicit in the assumptions and reasoning followed in M (*i.e.,* T does not go beyond M conceptually or in principle).

So???

The goal is to constructively attach programs to constructive real number as outlined in Bishop-Bridges, *Constructive Analysis*.

Within our program, we must meet Feferman's two criteria. The mathematical theory is constructive analysis.

Bootstrap Version

- Initial bootstrap version completed.
- Source to `C`
- Uses multi-precision number systems
- Supports both functional and relational models
- Is non-deterministic
- Use `succeed` and `fail` as control

The Constructive Real Numbers

A constructive real number is taken as a *regular sequence* of rational numbers. A sequence x is *regular* if

$$|x_m - x_n| \leq m^{-1} + n^{-1}.$$

[Bishop]

A *floating point* number is in the form $m \times b^s$ in the *Wilkinson set*

$$\pm m \times b^s \in \mathcal{W}(b, t, e_{min}, e_{max}).$$

Example Sun single-precision IEEE set $(2, 24, -125, 128)$.

Bootstrap Version—Virtual Machine Model

There are two separate considerations in the virtual machine: sequence control and data operations. Because of the dualist nature of the constructive model, we felt we needed both a functional and a relational ability.

The strong nature of the "fail" concept in constructive thinking is reminiscent of `Prolog` and `Snobol`. We therefore choose the Warren Abstract Machine as the sequence control virtual machine.

Bootstrap Version—Data Operational Model

The underlying data operational model for the bootstrap is multi-precision integers and rationals as provided by *GNU gmp* and multi-precision Wilkinson sets as provided by David Bailey's system.

Language Syntax

The language syntax is strongly influenced by current functional languages such as `haskell` but with major differences:

- Type syntax is not yet set. See below.
- The language uses Dijkstra's *guarded commands* as described Hesselink.
- Multiple abstractor types: λ for functional operation and π for relational operation.

Types.

Hindley-Milner type system seems counter to the strong constructivist basis. system. There is plenty to investigate:

- `Nuprl`

- **`Larch`**.

Integers, Rationals, and Floating (Wilkinson) plus strings.

Graph

Structured Data Types.

- Structured data types in the `struct` sense are available through an `assert` and unify mechanism.

- Vectors and matrices. *Organizations* are the key.

Subprogram Types.

The subprograms can be defined to follow a functional paradigm or a relational (but decidedly `Prolog`esque) paradigm. Anonymous functions can be developed using λ and π. Closures are also supported.

Sequence Control.

Sequence control is `Prolog` concept of *succeed-fail*. All operations whether in functional or relational mode either `succeed` or `fail`. The `if` and `do` statements are built using `try-retry-trust` in the WAM. Calls are made using `put-set`, while argument processing is done using `get-unify`. Call/return transfers are through a `dispatcher` (presaging parallel processing) and the use of `longjmp`S.

Storage Management. The storage management used is that of the WAM, plus two concessions to UNIX.