Operational Semantics in Languages and Logics

Robert Harper

Felleisen Halftime Festival
November 4, 2018
Way back when I first met Matthias, he was saying,

*I’m interested in all aspects of programming languages, except for types.*
Way back when I first met Matthias, he was saying,

*I’m interested in all aspects of programming languages, except for types.*

And I was saying,

*This guy is nuts!*
Way back when I first met Matthias, he was saying,

*I’m interested in all aspects of programming languages, except for types.*

And I was saying,

*This guy is nuts!*

And thinking,

*(Or is it me?)*
Way back when I first met Matthias, he was saying,

*I’m interested in all aspects of programming languages, except for types.*

And I was saying,

*This guy is nuts!*

And thinking,

*(Or is it me?)*

We seemed to be miles apart. Who is right?
It was the era of operational semantics.

- Plotkin’s *Structural Operational Semantics*.
- Milner, et al’s *Definition of Standard ML*.

And a hiatus for denotational semantics.

- Successful treatment of higher-order languages.
- How to account for effects?
It was also the era of *types for programming languages:*

- Milner’s type inference for ML.
- Reynolds’s and Girard’s theory of polymorphism.

And the era of *types for mathematics:*

- Martin-Löf: Intuitionistic Type Theory.
- Constable, Bates: NuPRL type theory and system.

We were just beginning to understand the now-famous correspondences.
Formal Propositions as Types

Formal logic (defined by rules):

\[ A_1 \text{ true, } \ldots, A_n \text{ true} \vdash A \text{ true} \]

Formal type systems (defined by rules):

\[ x_1 : A_1, \ldots, x_n : A_n \vdash M : A \]
Formal Propositions as Types

Formal logic (defined by rules):

\[ A_1 \text{ true, } \ldots, A_n \text{ true } \vdash A \text{ true} \]

Formal type systems (defined by rules):

\[ x_1 : A_1, \ldots, x_n : A_n \vdash M : A \]

By now this correspondence is true by definition!

- Not an isomorphism.
- Not solely due to Curry and Howard.
- No inherent computational meaning.
What Matthias Was Saying (Says Me)

What is the point of all these rules?

The subject matter is programming, done by people!

- Start with programs.
- Define types as specifications of behavior.

Emphasize meaning (content) over protocol (form).
What Matthias Was Saying (Says Me)

What is the point of all these rules?

The subject matter is programming, done by people!
  • Start with programs.
  • Define types as specifications of behavior.

Emphasize meaning (content) over protocol (form).

Where have we heard this before?
Contrary to Hilbert’s formalism, Brouwer proposed his intuitionism.

- Start with constructions performed by people.
- Define propositions as specifications of constructions.
Contrary to Hilbert’s formalism, Brouwer proposed his intuitionism.

- Start with constructions performed by people.
- Define propositions as specifications of constructions.

Mathematics is a human activity grounded in a shared understanding of construction.
Contrary to Hilbert’s formalism, Brouwer proposed his intuitionism.

- Start with constructions performed by people.
- Define propositions as specifications of constructions.

Mathematics is a human activity grounded in a shared understanding of construction. That is, programming is a human activity grounded in a shared understanding of computation.

Form follows function: formalisms are secondary matters of convenience.
Semantic Propositions as Types

Semantic Propositions as Types


Start with programs, define types as programs that classify other programs:

- A type means $A \Downarrow A_0$ and $A_0$ classifies other programs.
- $M \in A$ means $M \Downarrow M_0$ and $M_0$ is classified by $A_0$. 


Semantic Propositions as Types

Martin-Löf developed Brouwer’s ideas into a unified theory of computation and proof: 

_Constructive Mathematics and Computer Programming, 1982._

Start with programs, define types as programs that classify other programs:
- A type means $A \downarrow A_0$ and $A_0$ classifies other programs.
- $M \in A$ means $M \downarrow M_0$ and $M_0$ is classified by $A_0$.

More generally, define exact equality:
- $A \doteq B$ means $A \downarrow A_0$, $B \downarrow B_0$, and both classify the same programs.
- $M \doteq N \in A$ means $M \downarrow M_0$, $N \downarrow N_0$, and are interchangeable up to $A_0$. 
Voevodsky proposed the **univalence axiom** for formal type theory:

\[ \text{ua}(E) : \text{Equiv}(A, B) \simeq \text{Id}_U(A, B) \]

Equivalent types are identical (equivalence \( \simeq \) isomorphism).

**HoTT**: a **formal** account of univalence (and higher inductives):
- Add univalence as a **new axiom**.
- Justified by interpreting types as simplicial sets (spatial meaning).
The Univalence Axiom

Voevodsky proposed the univalence axiom for formal type theory:

$$\text{ua}(E) : \text{Equiv}(A, B) \simeq \text{Id}_U(A, B)$$

Equivalent types are identical (equivalence $\simeq$ isomorphism).

HoTT: a formal account of univalence (and higher inductives):

- Add univalence as a new axiom.
- Justified by interpreting types as simplicial sets (spatial meaning).

But what is the computational meaning of univalence?

$$J(x.N; \text{refl}_A(M)) \equiv N[M/x]$$

$$J(x.N; \text{ua}(E)) \equiv ???$$

Cannot just add axioms to type theory!
The difficulty is that $\text{Id}_A(M, N)$ is inductively defined:

- It is the least reflexive relation, universality witnessed by J.
- Defined uniformly in $A$!
The difficulty is that \( \text{Id}_A(M, N) \) is \textit{inductively defined}:

- It is the \textit{least reflexive} relation, universality witnessed by \( J \).
- Defined \textit{uniformly} in \( A \! 

Already there was trouble with \textit{function extensionality}:

\[
\text{Id}_{A \to B}(F, G) \not\simeq \forall x : A. \text{Id}_B(F x, G x).
\]

\text{Univalence is nothing but an extensionality principle.}
Crucially, the evidence matters . . . and what is evidence depends on the type!

\[
\text{ext}(P) \in \text{Path}_{A \rightarrow B}(F, G) \\
\text{ua}(E) \in \text{Path}_{U}(A, B)
\]
Identities and Paths

Crucially, the evidence matters . . . and what is evidence depends on the type!

\[ \text{ext}(P) \in \text{Path}_{A \rightarrow B}(F, G) \]
\[ \text{ua}(E) \in \text{Path}_{\_ U}(A, B) \]

Path types are equivalent, but not equal, to \text{Id} types.

- Not enough to provide computational meaning.
- No issue for the spatial meaning (modulo equivalence).
But what are path types? And how do we compute with them?

A path, or line, is a program parameterized by a dimension variable.

More generally, squares, cubes, hypercubes . . . .

Think of $x, y$ as continuously varying over $[0, 1]$ to draw the square.
Types are $n$-cubes: A type $[x_1, \ldots, x_n]$.

- An $n$-cube is a line between opposing faces in $n$ ways.
- $A \downarrow A_0$ and $A_0$ specifies $n$-cubes as elements.

Elements of $n$-cubes are $n$-cubes of elements: $M \in A [x_1, \ldots, x_n]$.

- $M \downarrow M_0$ and $M_0$ satisfies $A_0$ in all aspects.
- Coherently . . . (never mind) . . .

**Cubical operational semantics**: cubes compute their aspects.
There must be enough lines/squares/cubes of types and elements:

\[
\begin{array}{c}
\begin{array}{c}
\text{y} \\
\downarrow
\end{array}
\end{array}
\xymatrix{
\cdot \\
\downarrow^P
\ar[r]^M \\
\cdot \\
\downarrow_{\text{hcom}^{0\sim1}_A(M; y.P, y.Q)}
\downarrow^Q
\end{array}
\]

(Defines \(P^{-1} \cdot M \cdot Q\), combining inversion and concatenation.)
Type lines induce coercions between their end points:

\[ \text{coe}^{0 \sim 1}_{x \cdot A}(-) \in A\langle 0/x \rangle \to A\langle 1/x \rangle \ldots \]

whenever \( A \) is a line of types:

\[ A \in U \ldots , x \]

By univalence, all type lines are equivalences.

Coercion uses the equivalence to transport elements from one side to the other.

That is, the end points are interchangeable in all contexts!
Cubical proof assistants: \texttt{RedPRL}, \texttt{RedTT}.

\textit{Sterling, Favonia, Angiuli, Cavallo, Niu.}

Higher-dimensional parametricity: \texttt{relevance}.

\textit{Cavallo (cf, Vezossi)}

At Cornell, \texttt{free choice sequences}, the quintessential Brouwerian concept.

\textit{Bickford, Cohen, Constable, Rahli.}
There is a **science of computer programming** grounded in mathematics!

*Operational methods are fundamental, both conceptually and pedagogically.*
(Some of) What I Learned From Matthias

There is a science of computer programming grounded in mathematics!

Operational methods are fundamental, both conceptually and pedagogically.

Programming is a human activity: act as if people mattered!

Computation is about people first and foremost.
(Some of) What I Learned From Matthias

There is a science of computer programming grounded in mathematics!

*Operational methods are fundamental, both conceptually and pedagogically.*

Programming is a human activity: act as if people mattered!

*Computation is about people first and foremost.*

Stand up for what you believe in, you just may be right!

*(And don’t be afraid to be wrong.)*
Und wenn er nicht gestorben ist, dann lebt er noch heute!

Matthias is, of course, right!

(Except for the part about the types.)

And we were not that far apart after all:

Take computation seriously!
Matthias is, of course, right!

(Except for the part about the types.)

And we were not that far apart after all:

Take computation seriously!

Thank you, Matthias, for your influence, inspiration, irritation, and friendship.

And best wishes for the next half!
Thanks to many, including

- **Students**: Carlo Angiuli, Evan Cavallo, Favonia, Jon Sterling.
- **Colleagues**: Guillaume Brunerie, Thierry Coquand, Simon Huber, Dan Licata, Anders Mörtberg.
- **Inspiration**: Robert Constable, Per Martin-Löf, Vladimir Voevodsky†.

Supported by AFOSR MURI FA9550-15-1-0053, Tristan Nguyen, PM.


