The Verse Language: Types, Semantics, and Verification

Simon Peyton Jones

Lecture 4 - *July 2, 2026*

1 Denotational Semantics Revisited

1.1 Overview

While a rewrite (or operational) semantics uses syntactic transformations to define how a program might execute, a denotational semantics gives meaning (e.g. mathematical equivalences) to values and expressions.

Verse's novel design gives rise to three main challenges in defining its denotational semantics: (1) cardinality (a difficulty common to untyped languages), (2) the equivalence between patterns, expressions, and types, and (3) parametricity within Verse's nontraditional typing system. We address the first two challenges in this lecture.

2 Challenge 1: Cardinality Revisted

The following figure displays the simply typed lambda calculus (STLC) in Martin-Löf Type Theory. In this system, terms depend on types. Types and terms aren't distinct. Unlike in Verse.

Although the STLC provides a solution to the cardinality problem of the untyped λ-calculus as previously discussed, its requirement that every expression be typable poses a significant challenge for Verse. As shown in Figure 1, the semantic meaning function $\mathcal{E}$ requires a typing derivation of its input expression, making the STLC formulation incompatible with Verse.

Compiled By:
Anu Soneye, Bekah Sowards, Jack Phillips

$$
\begin{array}{rcl}
t & ::= & \mathit{Int} \mid t_1 \to t_2 \\
e & ::= & k \mid \mathit{op} \mid x \mid e_1\ e_2 \mid \lambda(x:t).\, e \\
\Gamma & ::= & \epsilon \mid \Gamma, x:t
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}[\![t]\!] & :: & \mathit{Type} \\
\mathcal{T}[\![\mathit{Int}]\!]\ \rho & = & \mathbb{Z} \\
\mathcal{T}[\![t_1 \to t_2]\!]\ \rho & = & \Pi(x : \mathcal{T}[\![t_1]\!]).\mathcal{T}[\![t_2]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}[\![\Gamma]\!] & :: & \mathit{dom}\ (\Gamma) \to \mathit{Type} \\
\mathcal{G}[\![\epsilon]\!] & = & \emptyset \\
\mathcal{G}[\![\Gamma, x:t]\!] & = & \mathcal{G}[\![\Gamma]\!], x :: \mathcal{T}[\![t]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![\Gamma \vdash e:t]\!] & :: & \mathcal{G}[\![\Gamma]\!] \to \mathcal{T}[\![t]\!] \\
\mathcal{E}[\![\Gamma \vdash k:\mathit{Int}]\!]\ \rho & = & k \\
\mathcal{E}[\![\Gamma \vdash x:t]\!]\ \rho & = & \rho(x) \\
\mathcal{E}[\![\Gamma \vdash (e_1\ e_2):t]\!]\ \rho & = & (\mathcal{E}[\![\Gamma \vdash e_1 : t_1 \to t]\!]\ \rho)(\mathcal{E}[\![\Gamma \vdash e_2 : t_1]\!]\ \rho) \\
 & & \text{where } \Gamma \vdash e_1 : t_1 \to t \text{ and } \Gamma \vdash e_2 : t_1 \\
\mathcal{E}[\![\Gamma \vdash (\lambda(x:t_1).\, e) : t_1 \to t_2]\!]\ \rho & = & \lambda\!\!\lambda(a \in \mathcal{T}[\![t_1]\!]).\, \mathcal{E}[\![\Gamma, x:t_1 \vdash e:t_2]\!]\ \rho[x := a]
\end{array}
$$

Figure 1: Simply Typed Lambda Calculus

The Verse team modified created a specification that eliminates the distinction between terms and types, as shown in Figure 2. In this formulation, there is no type system, and surprisingly utilizing Zermelo–Fraenkel (ZF) set theory alone is sufficient.

$$
\begin{array}{rcl}
t & ::= & \mathit{Int} \mid t_1 \to t_2 \\
e & ::= & k \mid \mathit{op} \mid x \mid e_1\ e_2 \mid \lambda(x:t).\, e
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{PI} & :: & \mathit{Set}(A) \to \mathit{Set}(A \times B) \to \mathit{Set}(A \to B) \\
\mathit{PI}(A, R) & = & \{f \mid f \subseteq R,\ \mathit{isFunction}(f),\ \mathit{dom}(f) = A\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}[\![t]\!] & :: & \mathit{Set} \\
\mathcal{T}[\![\mathit{Int}]\!]\ \rho & = & \mathbb{Z} \\
\mathcal{T}[\![t_1 \to t_2]\!]\ \rho & = & \mathit{PI}(\mathcal{T}[\![t_1]\!],\ \mathcal{T}[\![t_1]\!] \times \mathcal{T}[\![t_2]\!])
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![e]\!] & :: & \mathit{Env} \to \mathit{Set} \\
\mathcal{E}[\![k]\!]\ \rho & = & \{k\} \\
\mathcal{E}[\![x]\!]\ \rho & = & \{\rho(x)\} \\
\mathcal{E}[\![e_1\ e_2]\!]\ \rho & = & \{b \mid f \in \mathcal{E}[\![e_1]\!]\ \rho,\ a \in \mathcal{E}[\![e_2]\!]\ \rho,\ (a,b) \in f\} \\
\mathcal{E}[\![\lambda(x:t).\, e]\!]\ \rho & = & \mathit{PI}(A,\ \{(a,b) \mid a \in A,\ b \in \mathcal{E}[\![e]\!]\ (\rho[x := a])\}) \\
 & & \text{where } A = \mathcal{T}[\![t]\!] \\
\mathcal{E}[\![\mathbf{fix}\ e]\!]\ \rho & = & \{a \mid f \in \mathcal{E}[\![e]\!]\ \rho,\ (a,a) \in f\}
\end{array}
$$

Figure 2: Simply Typed Lambda Calculus using ZF Set Theory

Denotation of a type $\mathcal{T}[\![t]\!]$ is an (infinite) set of values that inhabit that type. To understand type conversions, $\mathcal{T}[\![t_1 \to t_2]\!]$, and the semantic meaning of $\mathcal{E}[\![\lambda(x:t).\, e]\!]\ \rho$, the PI auxiliary function is

fundamental. From set theory, a function is just a set of pairs. From the PI function seen in Figure 2, given a domain (a set A) and a set of pairs between the domain and range (a set B), PI returns all the functions from A to B (sets of pairs). Notice that the two inputs are sets, and the output is also a set.

Although STLC cannot express recursive functions, where there is a will, there is a way! You can easily define 'fix' in this system. Unlike the Simply Typed Lambda Calculus, which isn't nearly as expressive. Despite this, it's remarkably simple – even simpler than it because it just has terms and no types. It doesn't threaten the set-based semantics. Non-termination yields the empty set. Additionally, it doesn't find the least fixpoint, rather all fixpoints.

$$\mathcal{E}[\![\textbf{fix}\ e]\!]\ \rho\ =\ \{a\mid f\in\mathcal{E}[\![e]\!]\ \rho,\ (a,a)\in f\}$$

Figure 3: Timbda 0

2.0.1 Timbda 0

Timbda 0 is a preliminary primitive calculus for verse. Terms and types are the same thing, unlike the simply typed lambda calculus. In addition, there is only one lambda. While other calculi seperate terms and types, with different lambda terms, this calculus just has one. Timbda's only lambda term can build function types and functions.

$$\begin{array}{rcl}
e & ::= & k \mid \textbf{int} \mid x \mid e_1\ e_2 \\
& \mid & \lambda(x := e_1).\ e_2 \\
& \mid & :e \\
& \mid & \textbf{fail} \mid e_1 \mid e_2
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}[\![e]\!] & :: & Env \to Set \\
\mathcal{E}[\![k]\!]\ \rho & = & \{k\} \\
\mathcal{E}[\![x]\!]\ \rho & = & \{\rho(x)\} \\
\mathcal{E}[\![\textbf{int}]\!]\ \rho & = & \{\{(a,a)\mid a\in\mathbb{Z}\}\} \\
\mathcal{E}[\![:e]\!]\ \rho & = & \{b\mid f\in\mathcal{E}[\![e]\!]\ \rho,\ (a,b)\in f\} \\
\mathcal{E}[\![e_1\ e_2]\!]\ \rho & = & \{b\mid f\in\mathcal{E}[\![e_1]\!]\ \rho,\ a\in\mathcal{E}[\![e_2]\!]\ \rho,\ (a,b)\in f\} \\
\mathcal{E}[\![\lambda(x := e_1).\ e_2]\!]\ \rho & = & PI(A,\ \{(a,b)\mid a\in A,\ b\in\mathcal{E}[\![e_2]\!]\ (\rho[x := a])\}) \\
 & & \text{where } A = \mathcal{E}[\![e_1]\!]\ \rho \\
\mathcal{E}[\![\textbf{fail}]\!]\ \rho & = & \{\} \\
\mathcal{E}[\![e_1 \mid e_2]\!]\ \rho & = & \mathcal{E}[\![e_1]\!]\ \rho \cup \mathcal{E}[\![e_2]\!]\ \rho
\end{array}$$

Figure 4: Timbda 0

An expression denotes a set of values, like STLC. We need a set of results to express a type. So now the set is definitely NOT always a singleton or empty. It may have many, potentially infinite elements. However, it's still just ZF set theory, and there are no problems with cardinality. No type system: every term has a denotation. "Ill typed" terms simply denote the empty set.

3 Challenge 2: Unifying patterns and expressions

Expressions, given the value of free variables, output a value, while patterns, given an input argument, with the possibility of failing, return a set of bindings. Since Verse unifies patterns and expressions, the conventional denotation of an expression is insufficient for capturing pattern semantics. In particular, the standard denotation, $E[\![e]\!] : Env \to Val$, does not model the behavior of patterns: it neither takes an input argument to be matched nor produces the corresponding set of variable bindings.

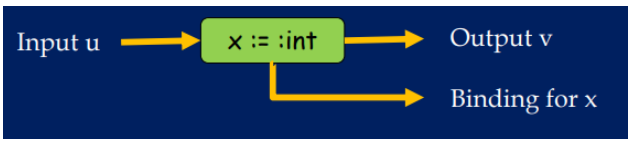

Figure 5: Unifying Expressions and Patterns

To account for this, we first define the semantics of expression–pattern unification as $E[\![e]\!]\, v : Ident \to Seq(ENV)$, where ENV represents an infinite set of environments ρ under specified constraints, specifically $ENV = SET(Env) = SET(Ident \to Val)$. The laws for this formulation are given in Figure 6. More generally, to account for patterns that match against an input value u, we define $E[\![e]\!]\, u\, v : Ident \times Ident \to Seq(ENV)$. The rules under this formalism can be seen on slide 43.

$$
\begin{aligned}
E[\![e]\!] &: \text{Ident} \to \text{Seq(ENV)} \\
E[\![\_]\!]v &= [\{\{\}\}] \\
E[\![k]\!]v &= [\{\{v = I(k)\}\}] \\
E[\![x]\!]v &= [\{\{v = x\}\}] \\
E[\![e_1 \mid e_2]\!]v &= E[\![e_1]\!]v + E[\![e_2]\!]v \\
E[\![\text{fail}]\!]v &= [] \\
E[\![e_1 = e_2]\!]v &= E[\![e_1]\!]v \ \otimes \ E[\![e_2]\!]v \\
E[\![e_1; e_2]\!]v &= (E[\![e_1]\!]q \setminus \{q\}) \ \otimes \ E[\![e_2]\!]v \\
E[\![x := e]\!]v &= [\{\{x = v\}\}] \ \otimes \ E[\![e]\!]v \\
E[\![: \text{int}]\!]v &= [\{\{v \in \mathbb{Z}\}\}]
\end{aligned}
$$

Figure 6: Denotational semantics of Verse expressions

4 Conclusion

This version of Verse, referred to as *MaxVerse*, is not yet used by engineers in production. Instead, they currently use *ShipVerse*, a language that is much closer to conventional object-oriented programming languages. The hope is that *MaxVerse* will either be adopted by programmers or influence the design of future programming languages!