The Verse Language: Types, Semantics, and Verification

Simon Peyton Jones

Lecture 3 - *July 1, 2026*

1 Big Picture

There are three languages: the surface language Source Verse, a subset of terms called Essential Verse, and a minimal subset called Core Verse. Source Verse is desugared into Essential Verse, which we can reason about using the denotational semantics or reduce to Core Verse, by means of matching, and apply the rewrite rules.

An Essential Verse program consists of terms and is therefore itself a term t. Core Verse, in contrast, consists of expressions. To translate Essential Verse into Core Verse, a fresh unification variable u (one that does not occur in t) is matched against t, written as $\exists u.\, u \rightsquigarrow t$. Essential Verse programs are translated into Core Verse because expressions are in a sense easier to 'run'. In particular, evaluation is made more explicit, such as how to apply β-reduction while respecting the scope of variable bindings.

2 Rewrite semantics, continued

2.1 Iter

Iter is a Core Verse primitive. It has three forms/combinators: all, if, and for. It's like a restricted version of the fold operation from other languages. This primitive can express the `if`, single-armed `for`, and double-armed `for` constructs in Essential Verse, which correspond to the if, all, and for combinators.

```
# Folding over SINGLE_ARMED FOR
for{1 | 2 | 3}
=> iter(all)(1 | 2 | 3){<>}
=> 1 'all' 2 'all' 3 'all' <>
```

Compiled By:
Anu Soneye, Bekah Sowards, Jack Phillips

```
=> <1,2,3>

# Folding over IF
# delay(e) approximately means  λx.e
if(x<7){x+1} else{3}
 => iter(if)(x<7;delay{x+1}){3} # assume x<7 does not fail
 => block{x+1}
```

$$
\begin{array}{lrcl}
\textsc{fail} & \textbf{iter}(ic)\{B[C[\textbf{fail}]]\}\{e\} & \longrightarrow & e \\
\textsc{choice} & \textbf{iter}(ic)\{B[C[b_1 \blacksquare b_2]]\}\{e\} & \longrightarrow & \textbf{iter}\ (ic)\{B[C[\textbf{block}\{b_1\}]]\} \\
 & & & \quad \{\textbf{iter}(ic)\{B[C[\textbf{block}\{b_2\}]]\}\{e3\}\} \\
 & & & \quad \text{if leftChoiceFree}(C) \\
 & & & \\
\textsc{if} & \textbf{iter}(\textbf{if})\{B[\textbf{delay}\{e_1\}]\}\{e_2\} & \longrightarrow & \textbf{block}\{B[e_1]\} \\
\textsc{all} & \textbf{iter}(\textbf{all})\{B[v]\}\{e\} & \longrightarrow & \textbf{cons}[B[v], e] \\
\textsc{for} & \textbf{iter}(\textbf{for})\{B[v]\}\{e\} & \longrightarrow & \textbf{block}\{\exists x\{\}.\ cons2[\langle x, B[v]\rangle, e]\}
\end{array}
$$

Figure 1: Iter Rules

2.2 Matching

Rewrite rules for matching an expression $u \sim\!> t$ match the pattern t against the input u, producing a binding for the binders in t, and resulting in failure if t does not match. The rules for matching each kind of expression follow this pattern. Written informal explanations for matching over various kinds of terms are paired with formal rewrite rules below.

For values (values, primitive operations, variables), matching is simply unification.

Unification

$$
\begin{array}{lrcll}
\textsc{eqk1} & k = k & \longrightarrow & k & \\
\textsc{eqk2} & k_1 = k_2 & \longrightarrow & \textbf{fail} & \text{if } k_1 \neq k_2 \\
\textsc{eqtup1} & \langle v_1, \cdots, v_n\rangle = \langle w_1, \cdots, w_n\rangle & \longrightarrow & v_1 = w_1;\ \cdots\ v_n = w_n;\ \langle v_1, \cdots, v_n\rangle & \\
\textsc{eqtup2} & \langle v_1, \cdots, v_n\rangle = \langle w_1, \cdots, w_m\rangle & \longrightarrow & \textbf{fail} & \text{if } n \neq m \\
\textsc{eqfail} & hnf_1 = hnf_2 & \longrightarrow & \textbf{fail} & \text{if } hnf_1 \text{ and } hnf_2 \text{ have different root shapes}
\end{array}
$$

Figure 2: Unification

For binding, the rewrite rules push 'u' into 't' and unify the bound name with the result.

$$u \rightsquigarrow_w x := t \longrightarrow x = (u \rightsquigarrow_w t)$$

Figure 3: MDEF

$$x \rightsquigarrow_w :t \longrightarrow \mathcal{D}_w[\![t]\!][x]$$

Figure 4: MCOLON1

Where 't' is of the form ':t1', the rewrite rules desugar 't1', then applies the result to 'u'.

Where 't' is of the form 't1;t2', the rewrite rules desugar 't1' and pushes 'u ~>' into 't2'.

$$u \rightsquigarrow_w t_1;\ t_2 \longrightarrow \mathcal{D}_w[\![t_1]\!];\ u \rightsquigarrow_w t_2$$

Figure 5: MSEMI

Where 't' is a function application, the rewrite rules apply the function, then unify the function result with 'u'.

$$u \rightsquigarrow_w t_1[t_2] \longrightarrow u = \mathcal{D}_w[\![t_1]\!][\mathcal{D}_w[\![t_2]\!]]$$

Figure 6: MAPP

Where 't' is unification of two terms, the rewrite rules push 'u' into both terms, then unifies the result.

$$u \rightsquigarrow_w t_1 = t_2 \longrightarrow (u \rightsquigarrow_w t_1) = (u \rightsquigarrow_w t_2)$$

Figure 7: MUNIF

Where 't' is a choice, the rewrite rules form scoping blocks for each choice, then pushes 'u' into each.

$$u \rightsquigarrow_w t_1 \mid t_2 \longrightarrow (u \rightsquigarrow_w \mathbf{block}\{t_1\}) \mid (u \rightsquigarrow_w \mathbf{block}\{t_2\})$$

Figure 8: MCHOICE

Where 't' is a function, the rewrite rules form scoping blocks for the argument and some additional variables. Then, pushes 'p' into the argument 'at', unifies, and applies the argument to the body 'bt'.When matching on a function, not only is static verification required, but also the unification algorithm is unable to synthesize a unification function that satisfies a relation between p and q. Consequently, $\mathcal{W}_o$, with an open circle, denotes the removal of the function h as a post-hoc solution.

$$h \rightsquigarrow_w \textbf{fun}(at)\{bt\} \quad \longrightarrow \quad \lambda u.\ \exists p, q, \mathsf{tbs}(at)\{\}.\ \ p = (u \rightsquigarrow_w at);\ \mathcal{W}_{bb}[\![q = h[p]]\!];\ q \rightsquigarrow_w \textbf{block}\{bt\}$$

where

$$\begin{aligned} (bb, dr, vx) &= w \\ \mathcal{W}_\circ[\![e]\!] &= \langle\rangle \\ \mathcal{W}_\bullet[\![e]\!] &= e \end{aligned}$$

Figure 9: MFUN

Where 'u' is matched against a tuple of terms, forming a block with scope over both $u_1 ... u_n$ and $y_1 ... y_n$. Because the tuple may contain type functions, we match each corresponding 'u' with 't', producing the transformed tuple $< y_1 ... y_n >$

$$u \rightsquigarrow_w \textbf{array}\{t_1, \cdots, t_n\} \longrightarrow \textbf{block}\{\exists\ u_1 \cdots u_n, y_1 \cdots y_n\{\}.\ u = \langle u_1, \cdots, u_n\rangle;\ y_1 = (u_1 \rightsquigarrow_w t_1); \cdots;\ y_n = (u_n \rightsquigarrow_w t_n);\ \langle y_1, \cdots, y_n\rangle\}$$

Figure 10: MTUP

3 Denotational Semantics

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 problems in defining its denotational semantics: cardinality (a difficulty common to untyped languages); the equivalence between patterns, expressions, and types; and parametricity within Verse's nontraditional typing system.

3.1 Cardinality

The cardinality problem arises from the difference between the size of the set of values that can be expressed by the abstract syntax and the size of the set of all functions over those values. Traditional approaches to solving the cardinality problem include domain theory, graph models, step-indexed models, and type theory. Each of these represents a different strategy to restrict the semantics

of the language to negate the cardinality problem. Each also has different incompatibilities with Verse's untyped system.

The Verse development team's plan for a denotational semantics uses a variation on the simply-typed lambda calculus to restrict the domain space of lambdas. Only the lambda binders have a type, ill-typedness is represented by empty set, and notably, the language itself remains free of types.

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

3.2 Future Work

Zooming out, a good rewrite system should have a sufficiently rich set of rules satisfying several key criteria such as being independently understandable, soundness with respect to the denotational semantics, and confluence. Consequently, future work includes:

1. Testing the rewrite system for confluence.
2. Formally proving confluence
3. Developing visualizations of the rewriting process.