The Verse Language: Types, Semantics, and Verification

Simon Peyton Jones

Lecture 2 - *June 30, 2026*

1 Matching

Verse unifies patterns, expressions and types; they share one semantics. Patterns are described using the squiggly arrow operator and abstraction. Neither is a part of the surface language. Here are some of the matching rules:

$$
\begin{aligned}
x : t \quad &\text{is shorthand for} \quad x := (: t) \\
f(at) := bt \quad &\text{is short for} \quad f := \mathit{function}(at)\{bt\} \\
i \sim\!> (x := t) &= x := (i \sim\!> t) \\
i \sim\!> (: t) &= t[i] \\
i \sim\!> k &= i = k \\
i \sim\!> (e_1, e_2) &= \exists j\, k.\ i = (j, k);\ j \sim\!> e_1;\ k \sim\!> e_2 \\
i \sim\!> (e_1; e_2) &= \exists j.\ (j \sim\!> e_1);\ (i \sim\!> e_2) \\
i \sim\!> (e_1 \mid e_2) &= (i \sim\!> e_1) \mid (i \sim\!> e_2) \\
i \sim\!> f[x] &= i = f[x] \\
i \sim\!> (e_1 \text{ where } e_2) &= \exists j.\ (i \sim\!> e_1);\ (j \sim\!> e_2)
\end{aligned}
$$

Interestingly, patterns can be functions.

```
# The domain of f is all functions mapping 1 to 3
f( function(1){3} ) := 7

# Applies abs to the argument, possibly failing, and transforming the argument
f := function( x := :abs ){x+2}
```

Compiled By:
Anu Soneye, Bekah Sowards, Jack Phillips

2 Rewriting

Rewrite rules define what a program written in the language does or means. They can be applied in any order and should (it's unproven) end in the same final result (confluence). In the example below, the first available rewrite rule (beginning from the left) is applied to successively transform the expression.

$$\exists x, y, r\{\}.\ x = 6; isInt[y]; r = (y, 77); y > 5; y = x + 3; r$$
$$\exists x, y, r\{x \leftarrow 6\}.\ 6; isInt[y]; r = (y, 77); y > 5; y = x + 3; r$$
$$\exists x, y, r\{x \leftarrow 6\}.\ isInt[y]; r = (y, 77); y > 5; y = x + 3; r$$
$$\exists x, y, r\{x \leftarrow 6, r \leftarrow (y, 77)\}.\ isInt[y]; (y, 77); y > 5; y = x + 3; r$$
$$\exists x, y, r\{x \leftarrow 6, r \leftarrow (y, 77)\}.\ isInt[y]; (y, 77); y > 5; y = 6 + 3; r$$
$$\ldots$$

2.1 Blocks

Blocks act as a primitive let, they allow you to add things to the heap. Specifically, it brings existential variables into scope, over a heap and body expression. Note, heap order does not matter.

$$\exists x, y\{x \leftarrow 3\}.\ y = x + 1$$

2.2 Unification

Unification is the algorithmic process of solving equations between symbolic expressions, each of the form Left-hand side = Right-hand side (Wiki). The Verse Calculus achieves unification through rewriting. It rewrites an expression until it evaluates to a value, fail, or it gets stuck (there are no rules left to apply). See the slides for the rules, but here are some examples.

```
1 = 1 # 1
1 = 2 # fail

(1, 2, 3) = (1, 2, 3) # (1, 2, 3)
(1, 2, 3) = (1, 2, 4) # fail

(x:=1) = (x:=1) # 1
(x:=1) = (x:=2) # fail
```

2.3 Heap and GC

Within a block, equalities are moved to the heap with the promotion rules.

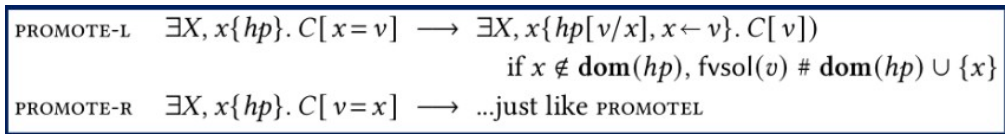

Figure 1: Promotion

After an equality has been promoted, the substitution rule substitutes the value for the variable one occurrence at a time in expressions enclosed within the block. The substitution context S allows for substitution deep within nested expressions (including lambda).

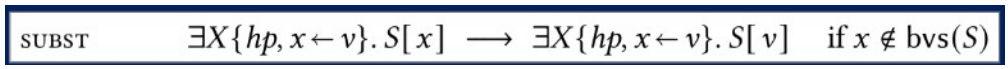

Figure 2: Substitution

After substitution, heap entries not mentioned in e that form strongly connected components can be removed by the garbage collection rule.

Figure 3: Garbage Collection