

## 1 Hierarchy of Lambda Calculus–Inspired Languages

We can organize the languages inspired by the lambda calculus as follows:

- Lambda calculus
  - Call-by-name lambda calculus
    - \* Haskell
    - \* Miranda
  - Call-by-value lambda calculus
    - \* (Lisp)
    - \* Scheme
    - \* ML

Notes:

- While the calculus was originally described by Church in the 1930s, it wasn't until Plotkin in 1975 that the distinction between call-by-name and call-by-value was made.
- Call-by-name means that  $(\lambda x.e)e_2 \simeq e[x := e_2]$  holds; under call-by-value,  $(\lambda x.e)v \simeq e[x := v]$  is true (and no less-restrictive statement about  $\simeq$  under  $\beta$ -reduction may be made).
- Lisp, while being inspired by the calculus, did not end up being a very sound implementation of it.

Given this taxonomy, we consider the question: how does each language support references and side effects?

**Haskell** Uses *monads*, a concept introduced in the 1990s.

**Miranda** Could similarly rely on monads.

**Scheme** Handles the problem “too aggressively.” Anything can be mutated with `(set! e)`.

**ML** Treats references as a unique entity, and they are the only objects in the language that can be mutated.

## 2 Semantics of References

We adopt a style similar to that of ML and define the following syntax:

$$e \in \mathbb{E} ::= \dots | \mathbf{ref} \ e | !e | e := e$$

We also need one more type:

$$T \in \mathbb{T} ::= \dots | T \ \mathbf{ref}$$

We define the typing rules as follows:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathbf{ref} \ e : T \ \mathbf{ref}} \quad \frac{\Gamma \vdash e : T \ \mathbf{ref}}{\Gamma \vdash !e : T}$$

$$\frac{\Gamma \vdash e_1 : T \ \mathbf{ref} \wedge \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : T \ \mathbf{ref}}$$

Notes:

- The rule for **ref** is the introduction rule; the rule for **!** is the elimination rule.
- If we didn't have an assignment operator but kept the first two rules, references would degenerate into nothing more than unary tuples or variants.
- The type of an assignment expression is somewhat arbitrary. We could have also chosen  $T$  or  $Unit$ .
- Unlike many language extensions, references don't have a nice Curry-Howard isomorphism with some construct in logic.

In order to define the evaluation rules, we have to redefine evaluation to carry with it a *heap*, which will act as a memory to which references may refer. Big-step evaluation is now a relation  $H, e \hookrightarrow H', e'$ . A heap is similar to  $\Gamma$ : a list of *label*-value pairs. A label is just an index value—the same restrictions that apply to the domain of variables apply to the domain of labels. We will insure that the heap always contains at most one value for each label.

The previous big-step rules are easily extended to adhere to our new definition:

$$\frac{}{H, \lambda x. e \hookrightarrow H, \lambda x. e} \quad \frac{\begin{array}{l} H_1, e_1 \hookrightarrow H_2, \lambda x. e_3 \\ H_2, e_2 \hookrightarrow H_3, e_4 \\ H_3, e_3[x := e_4] \hookrightarrow H_4, e_5 \end{array}}{H_1, e_1 \ e_2 \hookrightarrow H_4, e_5}$$

We now define the evaluation semantics of references as follows:

$$\begin{array}{c}
 \frac{H_1, e_1 \hookrightarrow H_2, e'_1}{H_3 \equiv H_2 : (l, e'_1)} \\
 \frac{H, e \hookrightarrow H', l}{H, !e \hookrightarrow H', e'} \\
 \frac{H_1, e_2 \hookrightarrow H_2, e'_2 \quad H_2, e_1 \hookrightarrow H_3, l \quad H_3 \equiv H_4 : (l, e_3) \quad H_5 \equiv H_4 : (l, e'_2)}{H_1, e_1 := e_2 \hookrightarrow H_5, l}
 \end{array}$$