Abstract

Plotkin's 1975 paper is strictly business. There are many theorems packed in the space of 35 pages, with little room for discussion, conclusions, and related/future work. This document provides a modern summary.

1 Historical Context

Plokin's *Call-By-Name, Call-By-Value, and the λ Calculus* [4] appeared 10 years after Peter J. Landin's *The Next 700 Programming Languages* [3], 11 years after the publication of the ALGOL-60 report, and 39 years after Alonzo Church published *An Unsolvable Problem of Elementary Number Theory* [1]. The significance of these works, in order of their appearance, was:

- Church's work introduced the λ calculus: a useful notation for reasoning about computable functions.
- The ALGOL-60 report gives a clean definition and semantics for a programming language. ALGOL-60 was implemented. You could run it on a variety of machines.
- Landin noticed a strong connection between languages like ALGOL-60 and the λ-calculus. His paper suggested using calculi to influence language design and gave a language called ISWIM to demonstrate.
- Plotkin formalized the correspondence between λ and ISWIM. Additionally, Plotkin proved one method of simulating a call-by-value λ-calculus with a call-by-name one, and other method in the opposite direction.

Offhand comment: Building connections is an important part of science, and especially computer science. The Church-Turing thesis unified different models of computation. This work by Plotkin unified language calculi with language implementations *and* by-name and by-value reduction semantics. Nowadays, the main source of connections is the propositions-as-types principle; we are slowly unifying type theory, logic, and category theory.

2 Results I: Lifting Computer Science

Before 1975, there were distinctly 2 kinds of PL researchers. Computer scientists built languages and proved properties of their implementation. Mathematicians worked in a formal system like the λ calculus and proved theorems that could ostensibly apply to any programming language. Plotkin elevated the meaning of "computer science" to include a possible formal system. Any theorems proven in the (below) calculus will definitely hold for the implementation.

2.1 ISWIM

Landin's programming language ISWIM is the starting point. This is a language we hope to reason about. ISWIM is defined using a set of basic constants, a function CONSTAPPLY from pairs of constants to terms, and the SECD abstract machine [2] (Figure 2.1).

Basic constants describe the core forms of the programming language, independent of the forms a programmer can define. These might include machine integers and I/O system calls. The special symbol **ap** is used internally by the machine's transition function, $\Rightarrow$.

$$
\begin{aligned}
\textit{Machine State} &= \langle S, E, C, D \rangle \\
\textit{Stack}(S) &= \textit{Closure}^* \\
\textit{Environment}(E) &= (\textit{Variable}, \textit{Closure})^* \\
\textit{Control}(C) &= (\textit{Term} \cup \mathbf{ap})^* \\
\textit{Dump}(D) &= \textit{Machine State}^*
\end{aligned}
$$

$$
\begin{aligned}
\textit{Closure}(Cl) &= \langle M, E \rangle \\
\textit{Constants} &= a, b, \ldots \\
\textit{Term}(M, N) &= x \mid a \mid \lambda x.\, M \mid M\ N \\
\textit{Variable} &= x, y, z, \ldots
\end{aligned}
$$

Side condition: for all closures $\langle M, E \rangle$, the free variables in the term M must be assigned values in the environment E.

$$
\begin{aligned}
\textit{Eval(M)} = N \iff \textit{Load}(M) \Rightarrow D \text{ and } \textit{Unload}(D) &= N \\
\textit{Load}(M) &= \langle \mathbf{nil}, \emptyset, M, \mathbf{nil} \rangle \\
\Rightarrow \quad &= \text{(omitted, see } \textbf{Section 3} \text{ of the paper)} \\
\textit{Unload}(\langle Cl, \emptyset, \mathbf{nil}, \mathbf{nil} \rangle) &= \textit{Real}(Cl)
\end{aligned}
$$

The function *Real* converts a closure to a term by replacing all free variables in a closure's term with corresponding values from the closure's environment.

Figure 1: The SECD machine. Sextiles (*) denote sequences.

This machine is what would run on the hardware; it is the low-level implementation of a programming language. Researchers could work directly with an implementation like this (and many do), but it is convenient to "elevate" the state of affairs and allow reasoning about an ISWIM interpreter:

$$
\begin{aligned}
\mathit{eval}(M) = N \iff & \exists t \,.\, M \Downarrow^t N \\
\textit{where} \quad & x \Downarrow x \\
& a \Downarrow a \\
& \lambda\, x.M \Downarrow \lambda\, x.M \\
& M\ N \Downarrow \textsc{ConstApply}(a, b) \\
& \qquad \text{if } \mathit{eval}(M) = a \text{ and } \mathit{eval}(N) = b \\
& M\ N \Downarrow [x/N']M' \\
& \qquad \text{if } \mathit{eval}(M) = \lambda\, x.\, M' \text{ and } \mathit{eval} N = N'
\end{aligned}
$$

Note: This interpreter uses a simple definition of substitution rather than the closure-and-environment model of the underlying machine. Also, evaluation will fail upon reaching a term undefined by $\Downarrow$.

The first theorem of the paper relates this mathematical *eval* to the SECD machine's *Eval*. Up to α-equivalence, of course.

Theorem. *For any program M, Eval*$(M) =_\alpha$ *eval(M).*

Proving this theorem requires the time index t from the definition of *eval*. The full proof is given in **Section 3**.

2.2 The λ_V Calculus

Beyond *eval*, it would be useful to work in Church's λ calculus to prove results about our programming language. At the very least, analogues of existing λ calculus theorems should hold for the language. In particular we should like Plotkin's theorem 4.4 to hold:

Theorem. *For any closed term M and value N,* $\text{Eval}(M) = N \iff \lambda \vdash M \geq N$

Here the symbol $\geq$ stands for "reduces to". In English, $\lambda \vdash M \geq N$ means there exists a proof using any λ rule but symmetry (i.e. α, β, transitivity) that $M = N$.

The plain λ calculus, however, is too free to directly correspond to *Eval*. It allows β reduction of arbitrary terms, does not include CONSTAPPLY, and allows reduction under a λ. Plotkin addresses these concerns by defining a λ_V calculus, wherein:

1. β-reduction may only substitute values.
2. Constants are values, and CONSTAPPLY is represented with Curry's notion of δ-reduction.
3. Reduction under a λ is avoided by defining a *standard reduction* order of evaluation and proving that any $\geq$ derivation implies a standard reduction derivation. The theorem then uses the first value along the standard reduction sequence as N.

Section 4 carries out this plan. The main proof effort is showing that $\geq$ implies a standard reduction sequence. That proof in turn requires λ_V be Church-Rosser (confluent).

After proving theorem 4.4, the section concludes with a defintion of *contextual equivalence* ($\simeq$) for *Eval* and proves:

Theorem. *If* $\lambda_V \vdash M = N$ *then* $M \simeq N$.

The converse does not hold, but at least all reasoning in λ_V holds for the *Eval* of the underlying programming language.

2.3 Reduction Semantics

In fact, the paper defines two version of *eval* and two λ calculi, subscripted by V and N. We have shown the call-by-value definitions, but Plotkin also gives a by-name version and states the same theorems connecting it to a by-name SECD machine. These theorems and their corollaries are given in **Section 5** of the paper.

3 Results II: Simulations

As a first example of reasoning about a concrete language using a more abstract λ calculus, Plotkin shows how to simulate λ_V by λ_N and vice-versa. Both simulations use *continuation-passing-style* (CPS) to make control flow explicit. In effect, this gives a CPS-converted term no choice but to evaluate in a determined order.

Following Plotkin, we first give the simulation of by-value reduction in a by-name language. For any term M, we compile to $\overline{M}$ as follows, using the reserved symbols κ,α, and β.

$$\begin{aligned}
\overline{x} &= \lambda\,\kappa.\,\kappa\;x \\
\overline{a} &= \lambda\,\kappa.\,\kappa\;a \\
\overline{\lambda\,x.\,M} &= \lambda\,\kappa.\,\kappa\;\lambda\,x.\,\overline{M} \\
\overline{M\;N} &= \lambda\,\kappa.\,\overline{M}(\lambda\,\alpha.\,\overline{N}(\lambda\,\beta.\,\alpha\beta\kappa))
\end{aligned}$$

By forcing application terms M and N into the head position before they are juxtaposed, Plotkin ensures that both reduce to a value they are applied.

The simulation of by-name using a by-value language is done via a conversion $\underline{M}$ using the term $I = \lambda\,x.\,x$. This conversion requires that constants be split between *functional constants* a and *basic constants* b. Functional constants are valid first arguments to CONSTAPPLY; basic constants are valid second arguments.

$$
\begin{aligned}
\underline{x} &= x \\
\underline{a} &= \lambda\,\kappa.\,\kappa(\lambda\,\alpha.\,a(\alpha\ I)) \\
\underline{b} &= \lambda\,\kappa.\,\kappa\ b \\
\underline{\lambda\,x.\,M} &= \lambda\,\kappa.\,\kappa(\lambda\,x.\,\underline{M}) \\
\underline{M\ N} &= \lambda\,\kappa.\,\underline{M}(\lambda\,\alpha.\,\alpha\underline{N}\kappa)
\end{aligned}
$$

Here the intuition is that arguments N are delayed under an administrative $\lambda\,\alpha$ and later applying a continuation triggers evaluation.

The main theorems of this section (**Section 6** in the paper) are simulation arguments. These use conversion functions Ψ and Φ to CPS-convert result values. This is necessary because the result of $\mathit{Eval}_V(\overline{M}\ I)$ may be an abstraction $\lambda\,x.\,\overline{M'}$. Thus we need to put the same administrative terms in the result of $\mathit{Eval}_N(M)$.

Theorem. $\Psi(\mathrm{Eval}_N(M)) = \mathrm{Eval}_V(\overline{M}\ I)$ *and* $\Phi(\mathrm{Eval}_V(M)) = \mathrm{Eval}_N(\underline{M}\ I)$

A second result shows that the compiled terms give the same result whether evaluted by-name or by-value:

Theorem. $\mathrm{Eval}_N(\overline{M}\ I) = \mathrm{Eval}_V(\overline{M}\ I)$ *and* $\mathrm{Eval}_V(\underline{M}\ I) = \mathrm{Eval}_N(\underline{M}\ I)$

Finally, the paper shows when one λ calculus may be used to reason about another.

Theorem. $\lambda_V \vdash M = N \Rightarrow \lambda_N \vdash \overline{M} = \overline{N}$ *and* $\lambda_N \vdash M = N \iff \lambda_V \vdash \underline{M} = \underline{N}$

Note that simulating by-value using a by-name language using this paper's techniques does not allow reasoning about the by-value λ calculus using the by-name λ calculus. On the other hand, simulating by-name using by-value requires that constants are classified as either functional or basic. So both directions have minor tradeoffs, in addition to the syntactic and operational burden of the extra λ-terms used to guide reduction order.

At any rate, the paper concludes with the proof of translation for a by-value implementing language.

4 Contributions

The long-standing contributions are:

- Programming languages and calculi must be defined *as pairs*.
- Using a CPS transformation, one can implement an evaluator for either by-name or by-value reduction independent of the semantics for the evaluator's language. The transformations do not give efficient code, but suffice for theoretical reasoning. Hypothetically, one could demonstrate a new feature for Haskell using a prototype built in OCaml.

References

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58, 1936.

[2] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[3] Peter J. Landin. The next 700 programming languages. 1966.

[4] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.