The Lambda Calculus and its Concurrent Execution:  
Lecture 17  
J. Meseguer

Further Reading

1. Basic Textbooks and the Lambda Calculus with Types

The original monograph by Alonzo Church, the creator of the \( \lambda \)-Calculus is:

- Alonzo Church, "The Calculus of Lambda-Conversion",  
  Annals of Mathematical Studies, 6, Princeton Univ. Press, 1941.

The most comprehensive text on the \( \lambda \)-Calculus has been written by Barendregt:


For many applications, typed versions of the \( \lambda \)-Calculus are very useful. Two excellent overviews of Typed \( \lambda \)-Calculi can be found in:

- H.P. Barendregt, "Lambda Calculi with Types," in S. Abramsky,  
  D.M. Gabbay and T. Maibaum, Handbook of Logic in Computer  

- H.P. Barendregt, W. Dekkers and R. Statman, "Lambda Calculus  
2. Substitution Calculi and Compilations to Combinators

As we have seen, substitution in the standard formulation of the \( \lambda \)-calculus is quite complex and hard to implement correctly. Therefore, an entire cottage industry of more efficient and easier to implement substitution calculi, or, as they are sometimes called explicit substitution calculi, have been proposed.

The original paper by de Bruijn is:


A good survey of de Bruijn notation and other topics can be found in:

- F. Kamareddine, "Reviewing the Classical and the de Bruijn Notation for \( \lambda \)-calculus and Pure Type Systems," J. Logic Computat. Vol. 11, No.3, 363-394, 2001

Several explicit substitution calculi have been proposed. Without trying to be exhaustive, the following can be mentioned:


Stehr is, to my knowledge, the most general such calculus, since it can be applied to any binding operator, not just to the λ-binder.

Explicit substitution calculi are, by themselves, a good option to implement λ-calculi. But the other alternative is, as we have already seen, compilation into combinators classically into classical CL combinators S, K and I. But many other possible choices of combinators and supercombinators are possible. An excellent book on this approach for compilation of functional languages is:


But there is still another possibility, namely, categorical combinators, where “categorical” is used in the sense of Category Theory. Really? Yes, really, and it very
practical. The key idea in the following: the most natural and beautiful models of typed lambda calculus are cartesian-closed categories. But such categories have a natural algebraic description by a (typed) signature $\Sigma_\lambda$ of operators. These are the categorical continuations.

This approach was used as the basis of an abstract machine, called the Categorical Abstract Machine, to compile the ML functional language into its CML implementation, which later expanded into the OCaml language.

The key paper is:


For a very detailed account of this entire approach, an excellent monograph is:


3. Parallel Functional Computation in the Lambda Calculus:

The Big Picture

These are of course many possible formalizations of the
\( \lambda \), \( \lambda - \eta \), and \( \lambda - \eta - \alpha \) calculus. We have seen the following:

1. Formalization of \( \lambda - B \delta i n \) its classical foundation as a rewrite theory (parametric)

\[
\mathbb{R}^{[N]}_{\lambda - \eta - \alpha}
\]

where \( \alpha \) is not executable

2. Formalization of the \( \lambda \)-calculus in de Bruijn notation, where \( \alpha \) becomes syntactic identity for closed lambda terms using de Bruijn numbers as a rewrite theory:

\[
\mathbb{R}^{DB}_{\lambda}
\]

3. Formalization of the \( \lambda \)-calculus Could be trivially extended to the \( \lambda - \eta - \gamma \)-calculus in CINNI notation, where \( \alpha \)-equivalence becomes syntactic identity for closed \( \lambda \)-terms adding an extra conditional equation that replaces all names by a fixed one, formalized as a rewrite theory

\[
\mathbb{R}^{CINNI}_{\lambda - [N]}
\]

Leaving aside the issue of \( \alpha \)-equivalence, all these rewrite theories have isomorphic computations. That is, for a concrete choice of a set \( \text{Vars} \) of variable names, we have isomorphisms of categories of concrete computation:
\[ \mathcal{T}_{\text{CL}}^{\text{cl}} \cong \mathcal{T}_{\text{DB}}^{\text{cl}} \cong \mathcal{T}_{\text{CMNN}}^{\text{cl}} \[ \mathcal{R}_{\alpha} \] \[ \text{Var} \] \]

where, to avoid the complexities of terms with free variables I have restricted the categories to the closed \( \lambda \)-terms. Note that the equation for \( \alpha \) is not executable in \( \mathcal{T}_{\lambda, \alpha} \[ \mathcal{R}_{\alpha} \] \[ \text{Var} \] \), but it is executable in \( \mathcal{T}_{\text{CMNN}} \[ \text{Var} \] \[ \lambda, \alpha \] \).

So, all these models give us a precise mathematical description of the parallel computations of functional programs expressed in the \( \lambda \)-calculus.

A natural question to ask [as we did for Petri nets] is: what are computations in these \( \lambda \)-categories like? That is, do they correspond to some already known descriptions of parallel \( \lambda \)-calculus computations?

The answer is yes! It has been given in:

The essential point of their answer is that, by adding a few additional equations \( E_{AD} \) to those defining \( \mathcal{C}_R[Vars] \) as a category, we obtain a quotient category \( \mathcal{C}_R[Vars]/E_{AD} \) that is isomorphic to the notion of permutation equivalence between parallel computations of the \( \lambda \)-calculus.