

Initial investigations into relating Logical and Computational Calculi

Roshan James

December 12, 2006

Contents

1	Introduction	1
I	Proof Theory	1
2	Natural Deduction and Sequent Calculus	1
2.1	Natural Deduction	1
2.2	Sequent Calculus	2
2.3	Proofs	2
2.3.1	Proving S using Natural Deduction	2
2.3.2	Proving S using Sequent Calculus	2
2.3.3	Simply Typed Lambda Calculus	3
2.3.4	Type Inference of S	3
3	Evaluation strategies: Call-By-Value and Call-By-Name	3
3.1	Abstract Machine for Call-by-Value evaluation	3
3.2	Abstract Machine for Call-by-Name evaluation	4
3.3	Abstract Machine for CBV	4
3.4	Abstract Machine for CBN	5
3.5	CPS	5
3.5.1	CBN CPS	5
3.5.2	CBV CPS	5
4	A Computational Calculus corresponding to Sequent Calculus	6
4.1	Introducing $\bar{\lambda}\mu\tilde{\mu}$	6
4.1.1	Term Language and Reductions of $\bar{\lambda}\mu\tilde{\mu}$	6
4.1.2	Translation of lambda calculus to $\bar{\lambda}\mu\tilde{\mu}$	6
4.2	CPS of $\bar{\lambda}\mu\tilde{\mu}$	7
4.2.1	CBV CPS of $\bar{\lambda}\mu\tilde{\mu}$	7
4.2.2	CBN CPS of $\bar{\lambda}\mu\tilde{\mu}$	7
5	Type theoretic foundations of $\bar{\lambda}\mu\tilde{\mu}$	7
5.1	Introducing $LK\mu\tilde{\mu}$	7
5.1.1	Rules for $LK\mu\tilde{\mu}$	7
5.1.2	Typing rules for $\bar{\lambda}\mu\tilde{\mu}$ using $LK\mu\tilde{\mu}$	8
5.2	Proofs in $LK\mu\tilde{\mu}$	8
5.2.1	Proving S via translation to $\bar{\lambda}\mu\tilde{\mu}$	8
II	Control Operations	10

6	CallCC	10
6.1	Abstract Machine for CallCC	10
6.2	Principal Type Scheme of CallCC: Pierce’s law	10
6.3	μ is similar to callcc	10
7	Other Control Operators	11
7.1	Abort: <i>ex falso quodlibet</i>	11
7.2	Felleisen’s <i>C</i> operator: double negation	11
7.3	CPS erases Control operations	11
8	Control Operators and Logics they induce	12
9	Delimited Continuations	12
10	The Call-By-Need Lambda Calculus	13
10.1	Abstract Machine by Zena Ariola et al	13
11	Call-by-Need using delimited continuations	14
12	Conclusions	14

1 Introduction

Note: Since this project is still work in progress, this document is written in the style of an informal report.

This report is split into two sections. The first of these investigates the relationship between natural deduction and sequent calculus from a computational perspective. The second section discusses control operators and their role in relating the principal type scheme of the lambda calculus to various logics and a formalization of the call-by-need lambda calculus using control operations.

It is hoped that by the time the work on this is completed one will be able to provide a formalization of call-by-need lambda calculus using a term language that is an extension of the lambda calculus with control and ‘environment’ operators. It would also be interesting to study what type/proof theoretic framework would be appropriate to type such a term language.

The work on this project started as a consequence of trying to formalize a CPS (Continuation-Passing-Style) translation for the call-by-need lambda calculus. Investigations into this topic shed light into several unexpected areas including the relationship between evaluation contexts and environments, sequent calculi and natural deduction etc. I hope to present some aspects of this study in the form of this informal report.

Being work in progress it is highly likely that much of the information here maybe incomplete, potentially incorrect or have been studied earlier. Also this report lacks any single unifying result to present, though the presentation of call-by-need using control operations and investigations into the completeness of $LK\mu\tilde{\mu}$ might merit some attention. Most of this report however consists of a survey of ideas that I expect are relevant to the expected results stated previously.

Part I

Proof Theory

2 Natural Deduction and Sequent Calculus

The Curry-Howard Isomorphism relates propositions to types, proof derivations under natural deduction to terms and proof normalization to evaluation. Typing judgements for terms are usually specified in a hybrid style that is sometimes described as “sequent style natural deduction”. Sequent style natural deduction has the advantage that the assumptions of derivation tree are always present in the form on a environment Γ . However it is not obvious how one may specify the typing judgements of the lambda calculus in a sequent calculus.

2.1 Natural Deduction

The following are the rules for the implicational fragment of intuitionistic logic in Gentzen's Natural deduction [4], as formalized by Prawitz [10]:

$$\frac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \rightarrow B} \rightarrow I \quad \frac{A \quad A \rightarrow B}{B} \rightarrow E$$

2.2 Sequent Calculus

The following are the rules for the implicational fragment of intuitionistic logic in Gentzen's sequent calculus [4] [10]:

$$\frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B} \rightarrow R \quad \frac{\Gamma \vdash \Delta_1, A \quad B, \Gamma \vdash \Delta_2}{A \rightarrow B, \Gamma \vdash \Delta_1 \cup \Delta_2} \rightarrow L$$

2.3 Proofs

Here is a proof of the principal type scheme of combinator S in both proof theoretic systems.

$$S :: (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$$

2.3.1 Proving S using Natural Deduction

$$\frac{\frac{\frac{a \rightarrow b \quad a}{b} \rightarrow E \quad \frac{a \quad a \rightarrow (b \rightarrow c)}{b \rightarrow c} \rightarrow E}{\frac{c}{a \rightarrow c} \rightarrow I(a)} \rightarrow I(a \rightarrow b)}{(a \rightarrow b) \rightarrow (a \rightarrow c)} \rightarrow I(a \rightarrow (b \rightarrow c))$$

2.3.2 Proving S using Sequent Calculus

$$\frac{\frac{\frac{a, b \vdash b \quad c, a, b, \vdash c}{b \rightarrow c, a, b \vdash c} \rightarrow L \quad \frac{b \rightarrow c, a \vdash a}{b \rightarrow c, a \rightarrow b, a \vdash c} \rightarrow L}{\frac{a \rightarrow b, a \vdash a}{a \rightarrow (b \rightarrow c), a \rightarrow b, a \vdash c} \rightarrow R} \rightarrow R}{\frac{a \rightarrow (b \rightarrow c), a \rightarrow b \vdash a \rightarrow c}{a \rightarrow (b \rightarrow c) \vdash (a \rightarrow b) \rightarrow (a \rightarrow c)} \rightarrow R} \rightarrow R$$

My initial approach to writing proofs using these proof theoretic formulations was to look at them as programming languages and proofs as programming tasks. Indeed the Curry-Howard isomorphism indicates that derivation trees of proof deduction correspond to programs. Using this as the intuition for approaching proofs of propositions it seemed easy to prove propositions under natural deduction. However to prove the same proposition under the sequent calculus felt much harder and often times non-obvious.

The interesting observation was that there seemed to be familiar feeling about this difficulty in doing proofs using the sequent calculus. In some senses it seemed much like the difficulty of turning a recursive style program into an accumulator passing style program (for example. doing tree traversal without using recursion) - better described as trying to write programs in a CPS style as opposed to direct style.

Looking at the proof of a proposition in natural deduction, one can extract the corresponding term in lambda calculus. The converse of this is usually specified as typing judgements. Provided below are the typing judgements of the simply typed lambda calculus. The assumptions of the derivation tree are 'collected into a sequent'. Hence description 'sequent style natural deduction' seems appropriate for such typing derivations[12].

2.3.3 Simply Typed Lambda Calculus

Expressions:

$$e = x \mid \lambda x.e \mid e_1 e_2$$

Type Judgements:

$$\frac{}{\Gamma, x : t \vdash x : t} \text{var} \quad \frac{\Gamma, (x : t_1) \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \rightarrow t_2} \text{lam} \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1 e_2) : t_2} \text{app}$$

If the term under type reconstruction has no shadowed variables then one may use the formalization of Γ as a set where each element is a variable name annotated with a proposition that its principal type. This annotation is indicated using the syntactic ‘:’.

2.3.4 Type Inference of S

The S combinator maybe expressed as the following lambda expression:

$$S = \lambda x.\lambda y.\lambda z.((x z) (y z))$$

Here is the type inference of S :

$$\frac{\frac{\frac{\frac{\Gamma' \vdash x : a \rightarrow (b \rightarrow c)}{\Gamma' \vdash (x z) : b \rightarrow c} \text{var} \quad \frac{\Gamma' \vdash z : a}{\Gamma' \vdash (y z) : b} \text{var}}{\Gamma' \vdash ((x z) (y z)) : c} \text{app} \quad \frac{\Gamma' \vdash y : a \rightarrow b \quad \Gamma' \vdash z : a}{\Gamma' \vdash (y z) : b} \text{var}}{\Gamma' \vdash ((x z) (y z)) : c} \text{app} \quad \frac{\Gamma, x : (a \rightarrow (b \rightarrow c)), y : (a \rightarrow b) \vdash \lambda z.((x z) (y z)) : a \rightarrow c}{\Gamma, x : (a \rightarrow (b \rightarrow c)) \vdash \lambda y.\lambda z.((x z) (y z)) : (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{lam}}{\Gamma \vdash \lambda x.\lambda y.\lambda z.((x z) (y z)) : (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))} \text{lam}$$

where

$$\Gamma' = \Gamma, x : (a \rightarrow (b \rightarrow c)), y : (a \rightarrow b), z : a$$

A proof derivation using the sequent calculus without the cut rule is always normal. In Prawitz’s text on Natural Deduction he makes the observation in [10] that one may look a proof in the sequent calculus as ‘guiding a proof in natural deduction’. Prawitz goes on to further note that there maybe many normal proofs under the natural deduction corresponding to the same derivation in the sequent calculus. This seemed reminiscent of a computational analogy, the CPS translation fixes the evaluation semantics of a lambda term.

3 Evaluation strategies: Call-By-Value and Call-By-Name

3.1 Abstract Machine for Call-by-Value evaluation

Call-by-value evaluation is described by the abstract machine below.

Syntactic Categories:

$$\begin{aligned} \text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \\ \text{Syntactic Values, } v &= x \mid \lambda x.e \\ \text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\ \\ \text{Runtime Values, } w &= \langle \text{closure } \rho \ x \ e \rangle \\ \text{Environments, } \rho &= \square \mid (x = v) : \rho \\ E &= \square \mid (\square \ e \ \rho) : E \mid (w \ \square) : E \end{aligned}$$

Reduction Rules:

$$\begin{aligned}
\langle e_1 e_2, \rho, E \rangle &\mapsto \langle e_1, \rho, (\square e_2 \rho) : E \rangle \\
\langle v, \rho, (\square E) \rangle &\mapsto \langle \phi(v, \rho), \rho, (\square e_2) : E \rangle \\
\langle w, \rho', (\square e \rho) : E \rangle &\mapsto \langle e, \rho, (w \square) : E \rangle \\
\langle w, \rho', (\langle \text{cls } \rho x e \rangle \square) : E \rangle &\mapsto \langle e, (x = w) : \rho, E \rangle \\
\phi(\lambda x.e, \rho) &= \langle \text{cls } \rho x e \rangle \\
\phi(x, \rho) &= \rho(x)
\end{aligned}$$

Programs are evaluated by starting the machine with the initial state $\langle e, \square, \square \rangle$ where e is the program to be evaluated. Evaluation is said to be complete when the machine reaches the state $\langle w, \rho, \square \rangle$ with the result w . The reflective transitive closure \mapsto^* of the reduction relation \mapsto can be used to construct a partial function from terms to runtime values that corresponds to a call-by-value interpreter for the term language.

3.2 Abstract Machine for Call-by-Name evaluation

Call-by-name evaluation is described by the abstract machine below.

Syntactic Categories:

$$\begin{aligned}
\text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \\
\text{Syntactic Values, } v &= x \mid \lambda x.e \\
\text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\
\text{Runtime Values, } w &= \langle \text{closure } \rho x e \rangle \\
\text{Environments, } \rho &= \square \mid (x = (e, \rho_1)) : \rho_2 \\
E &= \square \mid (\square e \rho) : E
\end{aligned}$$

Reductions

$$\begin{aligned}
\langle e_1 e_2, \rho, E \rangle &\mapsto \langle e_1, \rho, (\square e_2 \rho) : E \rangle \\
\langle \langle \text{closure } \rho x e \rangle, \rho', (\square e' \rho') : E \rangle &\mapsto \langle e, (x = (e', \rho')) : \rho, E \rangle \\
\langle x, \rho, E \rangle &\mapsto \langle e', \rho', E \rangle \text{ where } \rho(x) = (e', \rho') \\
\langle \lambda x.e, \rho, E \rangle &\mapsto \langle \langle \text{closure } \rho x e \rangle, \rho, E \rangle
\end{aligned}$$

The essential difference in the CBN machine is that bindings in the environment are pairs of unevaluated expressions and the environments in which they should be evaluated. Application of a closure to any expression causes the expression do be repeatedly evaluated as many times as the body of the closure encounters its bound variable during its evaluation.

The abstract machine above defines an interpreter for the CBN lambda calculus in exactly the same way as for the CBV machine.

It has been shown by Plotkin [9] and others that call-by-value and call-by-name induce different equational theories, ie the number of reduction steps, termination of evaluation etc are different under both evaluation strategies.

3.3 Abstract Machine for CBV

The following is a more concise formulation of the operational semantics of CBV lambda calculus. This specification essentially differs from the previous one in the sense that it use tree-structured evaluation contexts as formalized by Matthias Felleisen[3]. The specification of the evaluation context guides the search for redexes in the current expression.

Syntactic Categories:

$$\begin{aligned}
\text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \\
\text{Syntactic Values, } v &= \lambda x.e \\
\text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\
\text{Evaluation Contexts, } E &= \square \mid v E \mid E e
\end{aligned}$$

It can be shown that any given expression e can be uniquely decomposed into an expression and an evaluation context. The decomposition is guided by the definition of the syntactic category of evaluation contexts. Such

a decomposition is syntactically represented as $E[e']$, where E is the context and the e' is the sub-expression. Reduction rules in such a system are rewrite rules on context-expression pairs.

Any abstract machine represented in this fashion can be rewritten as a set of rewrite rules on tuples that make code, environment and stack changes explicit like the previous abstract machine definitions.

Reductions:

$$E[(\lambda x.e) v] \mapsto E[e[v/x]]$$

This machine requires only one reduction rule which is essentially the beta rule of the lambda calculus. The syntax $e[v/x]$ represent the syntactic substitution of every free variable x in e with v . Since this machine relies on substitution there is no need for an explicit environment.

I introduce this formalization of abstract machines simply for the brevity it offers. This brevity will be useful later on in the paper when looking at control operations and their semantics.

3.4 Abstract Machine for CBN

For completeness I present a similar abstract machine for CBN evaluation.

Syntactic Categories:

$$\begin{aligned} \text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \\ \text{Syntactic Values, } v &= \lambda x.e \\ \text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\ \text{Evaluation Contexts, } E &= \square \mid E e \end{aligned}$$

Reductions:

$$E[(\lambda x.e_1) e_2] \mapsto E[e_1[e_2/x]]$$

The reduction rule above brings out the essential difference in CBN which is that the operand is not evaluated but is instead substituted into the operators body. This causes the operand to be evaluated for every occurrence of the bound variable encountered during the evaluation of e_1 .

3.5 CPS

The evaluation of a lambda term is completely specified by the providing its CPS translation. Corresponding to call-by-value and call-by-name evaluation there are two CPS translations.

3.5.1 CBN CPS

The CBN CPS translation as formalized by Plotkin [9] and refined by Hatcliff and Danvy [5] [1]:

$$\begin{aligned} [x] &= x \\ [\lambda x.e] &= \lambda k.(k \lambda x.[e]) \\ [e_1 e_2] &= \lambda k.([e_1] \lambda f.(f [e_2] k)) \end{aligned}$$

3.5.2 CBV CPS

The CBV CPS translation as formalized by Plotkin [9]:

$$\begin{aligned} [x] &= \lambda k.(k x) \\ [\lambda x.e] &= \lambda k.(k \lambda x.[e]) \\ [e_1 e_2] &= \lambda k.([e_1] \lambda f.(f [e_2] \lambda v.(f v k))) \end{aligned}$$

One may understand the phrase ‘CPS fixes the evaluation strategy’ to mean that the CPS translated terms evaluate exactly the same way in both the abstract machines provided above. It is also interesting to note that the ‘current context’ during the evaluation of a CPSed term is always the empty context i.e. \square .

4 A Computational Calculus corresponding to Sequent Calculus

4.1 Introducing $\bar{\lambda}\mu\tilde{\mu}$

In this context the $\bar{\lambda}\mu\tilde{\mu}$ [2] [11] is worthy of investigation. The $\bar{\lambda}\mu\tilde{\mu}$ calculus formalized by Curien and Herberlin and is a computational calculus whose type theoretic foundations are based on the sequent calculus. The following is the term language:

4.1.1 Term Language and Reductions of $\bar{\lambda}\mu\tilde{\mu}$

The term language of the $\bar{\lambda}\mu\tilde{\mu}$ calculus involves three syntactic categories called terms, contexts and commands (which are essentially composition of terms and contexts).

Syntactic Categories:

$$\begin{aligned} \text{Terms, } e &= \mu\alpha.c \mid \lambda x.e \mid x \\ \text{Contexts, } E &= \tilde{\mu}x.c \mid \alpha \mid e.E \\ \text{Commands, } c &= \langle e \mid E \rangle \end{aligned}$$

Reductions:

$$\begin{aligned} \langle \lambda x.e \mid v.E \rangle &\mapsto \langle e[v/x] \mid E \rangle \\ \langle \mu\alpha.c \mid E \rangle &\mapsto c[E/\alpha] \\ \langle e \mid \tilde{\mu}x.c \rangle &\mapsto c[e/x] \end{aligned}$$

Reductions apply only to ‘commands’ in this calculus. The $\bar{\lambda}\mu\tilde{\mu}$ calculus is not confluent.

4.1.2 Translation of lambda calculus to $\bar{\lambda}\mu\tilde{\mu}$

The following is a translation of lambda calculus to the $\bar{\lambda}\mu\tilde{\mu}$ calculus (i.e. $\lambda \rightarrow \bar{\lambda}\mu\tilde{\mu}$). An interesting aspect of the translation is that it is type preserving.

$$\begin{aligned} [x] &= x \\ [\lambda x.e] &= \lambda x.[e] \\ [e_1 e_2] &= \mu\alpha.\langle [e_2] \mid \tilde{\mu}x.\langle [e_1] \mid x.\alpha \rangle \rangle \end{aligned}$$

What is it that the $\bar{\lambda}\mu\tilde{\mu}$ does? To be perfectly honest, I do not understand it completely yet. But the intuition is that $\bar{\lambda}\mu\tilde{\mu}$ exposes a duality between terms under evaluation and their environment i.e. this duality is expressed used a syntax that makes this symmetry apparent. The translation of lambda terms to $\bar{\lambda}\mu\tilde{\mu}$ is reminiscent of the CPS translations previously introduced, but it is not exactly equivalent to either of them. It seems as if the one has stopped midway between the translation of a lambda term to its CPS equivalent under a particular evaluation strategy.

Indeed this intuition gives us some mileage because it is in agreement with the key contribution of Curien and Herbelin’s [2] paper which is that call-by-name and call-by-value are duals of each other. This duality is presented by them as choice of precedence between μ and $\tilde{\mu}$. When the evaluator faces the choice of reducing μ and $\tilde{\mu}$, it chooses between call-by-value and call-by-name reduction.

$$\begin{aligned} \langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle &\mapsto_{cbv} c_1 [\tilde{\mu}x.c_2/\alpha] \\ \langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle &\mapsto_{cbn} c_2 [\mu\alpha.c_1/x] \end{aligned}$$

So in some vague sense one may conclude that the translation of lambda terms to $\bar{\lambda}\mu\tilde{\mu}$ has some aspects of a CPS translation but omits the aspects that commit to an evaluation strategy. I hope to make these intuitions more formal with future work in this area.

This translation has also caused type derivation for the calculus to change from natural deduction to sequent calculus. I have unsuccessfully attempted to formalize call-by-need evaluation using the μ and $\tilde{\mu}$ operators of this calculus. Call-by-need appears to fall part way between call-by-value and call-by-name evaluation. While there is no proof to the fact that it is not possible, the fact that there is no obvious formalization of call-by-need

using $\bar{\lambda}\mu\tilde{\mu}$ leads me to suspect that the one may need additional operators and potentially a structural change in the $\bar{\lambda}\mu\tilde{\mu}$ calculus to accommodate the formalization of call-by-need. What changes will this induce on the type theoretic formalization of such a calculus? What will this reveal about natural deduction and sequent calculus? This is another unexplored area.

4.2 CPS of $\bar{\lambda}\mu\tilde{\mu}$

Curien and Herberlin provide two CPS translations of the $\bar{\lambda}\mu\tilde{\mu}$ calculus to lambda terms that correspond to the choice of precedence between μ and $\tilde{\mu}$ - hence correspond to a CBV and CBN translations.

In the CPS translations provided below I have taken the liberty of some simplifications to the CPS presented in the original paper.

4.2.1 CBV CPS of $\bar{\lambda}\mu\tilde{\mu}$

$$\begin{aligned} [(e \mid E)]^\triangleleft &= [e]^\triangleleft [E]^\triangleleft \\ [\alpha]^\triangleleft &= \alpha \\ [x]^\triangleleft &= \lambda k.(k \ x) \\ [\mu\alpha.c]^\triangleleft &= \lambda\alpha.[c]^\triangleleft \\ [\tilde{\mu}x.c]^\triangleleft &= \lambda x.[c]^\triangleleft \\ [\lambda x.e]^\triangleleft &= \lambda k.(k \ \lambda x.[e]^\triangleleft) \\ [e.E]^\triangleleft &= \lambda k.([e]^\triangleleft \ \lambda v.(k \ v \ [E]^\triangleleft)) \end{aligned}$$

4.2.2 CBN CPS of $\bar{\lambda}\mu\tilde{\mu}$

$$\begin{aligned} [(e \mid E)]^\triangleright &= [e]^\triangleright [E]^\triangleright \\ [\alpha]^\triangleright &= \lambda k.(k \ \alpha) \\ [x]^\triangleright &= x \\ [\mu\alpha.c]^\triangleright &= \lambda\alpha.[c]^\triangleright \\ [\tilde{\mu}x.c]^\triangleright &= \lambda x.[c]^\triangleright \\ [\lambda x.e]^\triangleright &= \lambda k.\lambda x.([e]^\triangleright \ k) \\ [e.E]^\triangleright &= \lambda k.([E]^\triangleright \ \lambda v.(k \ v \ [e]^\triangleright)) \end{aligned}$$

5 Type theoretic foundations of $\bar{\lambda}\mu\tilde{\mu}$

5.1 Introducing $LK\mu\tilde{\mu}$

The implicational fragment of classical logic presented in a sequent calculus called $LK\mu\tilde{\mu}$ provides the type judgements for $\bar{\lambda}\mu\tilde{\mu}$. The $LK\mu\tilde{\mu}$ differs from Gentzen's LK in that it has the notion of a special focused term. Proof deductions can progress only on the basis of what term is currently in focus. It is claimed that the notion of a focused term limits the proof search such that there is only one normal form proof corresponding to a proposition [12].

5.1.1 Rules for $LK\mu\tilde{\mu}$

The focused term in the rules below is the one closest to the \vdash . We use the $|$ to distinguish the focused term from other terms in the antecedent or the consequent. The $|$ is reminiscent of the $;$ separation operator that is used in the formalization of sequent calculi for relevance logics.

$$\begin{array}{c} \frac{}{\Gamma \mid A \vdash A, \Delta} \text{AxL} \qquad \frac{}{\Gamma, A \vdash A \mid \Delta} \text{AxR} \\ \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \rightarrow B \vdash \Delta} \rightarrow L \qquad \frac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \rightarrow B \mid C} \rightarrow R \\ \frac{\Gamma, A \vdash \Delta}{\Gamma \mid A \vdash \Delta} \text{ActivateL} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A, \Delta} \text{ActivateR} \end{array}$$

$$\frac{\Gamma \vdash A \mid \Delta \quad \Gamma \mid A \vdash \Delta}{\Gamma \vdash \Delta} \text{Cut}$$

The logic $LK\mu\tilde{\mu}$ can be annotated with terms of $\bar{\lambda}\mu\tilde{\mu}$ providing the type theoretic foundation for $\bar{\lambda}\mu\tilde{\mu}$.

5.1.2 Typing rules for $\bar{\lambda}\mu\tilde{\mu}$ using $LK\mu\tilde{\mu}$

$$\begin{array}{c} \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta} \text{AxL} \qquad \frac{}{\Gamma, x : A \vdash a : A \mid \Delta} \text{AxR} \\ \\ \frac{\Gamma \vdash e : A \mid \Delta \quad \Gamma \mid E : B \vdash \Delta}{\Gamma \mid e.E : A \rightarrow B \vdash \Delta} \rightarrow L \qquad \frac{\Gamma, x : A \vdash e : B \mid \Delta}{\Gamma \vdash \lambda x.e : A \rightarrow B \mid C} \rightarrow R \\ \\ \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \text{ActivateL} \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A, \Delta} \text{ActivateR} \\ \\ \frac{\Gamma \vdash e : A \mid \Delta \quad \Gamma \mid E : A \vdash \Delta}{\langle e \mid E \rangle : (\Gamma \vdash \Delta)} \text{Cut} \end{array}$$

5.2 Proofs in $LK\mu\tilde{\mu}$

While it is not easy to see how implicational propositions can be proved directly in $LK\mu\tilde{\mu}$, the logic's correspondence to the typing judgements of the $\bar{\lambda}\mu\tilde{\mu}$ calculus enables us to leverage the Curry-Howard Isomorphism to show its completeness.

A proof corresponding to a classical implicational proposition can be found by first type-inhabiting the proposition using the lambda calculus. The resulting lambda calculus expression can be translated to the corresponding $\bar{\lambda}\mu\tilde{\mu}$ term using the translation previously provided. The interesting aspect of this translation is that it is type preserving i.e. the result $\bar{\lambda}\mu\tilde{\mu}$ term will have the same type. Hence type inferencing the term using the typing rules for $\bar{\lambda}\mu\tilde{\mu}$ will result in a derivation tree that is essentially a proof of the proposition in $LK\mu\tilde{\mu}$.

In some sense this process maybe considered a way to translate a proof in natural deduction to a proof in sequent calculus. Note that this valuable only to show that a proof exists in $LK\mu\tilde{\mu}$ - type habitation by itself corresponds to a proof in natural deduction.

5.2.1 Proving S via translation to $\bar{\lambda}\mu\tilde{\mu}$

The following is the principal type and the lambda term of the combinator S :

$$\begin{aligned} S &:: (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)) \\ S &= \lambda x.\lambda y.\lambda z.((x z) (y z)) \end{aligned}$$

The following is a step by step translation of S to $\bar{\lambda}\mu\tilde{\mu}$:

$$\begin{aligned} &\Rightarrow [\lambda x.\lambda y.\lambda z.((x z) (y z))] \\ &\Rightarrow \lambda x.\lambda y.\lambda z. [((x z) (y z))] \\ &\Rightarrow \lambda x.\lambda y.\lambda z.\mu\alpha_1. \langle [(y z) \mid \tilde{\mu}x_1. \langle [(x z)] \mid x_1.\alpha_1 \rangle \rangle \\ &\Rightarrow \lambda x.\lambda y.\lambda z.\mu\alpha_1. \langle \mu\alpha_2. \langle z \mid \tilde{\mu}x_2. \langle y \mid x_2.\alpha_2 \rangle \rangle \mid \tilde{\mu}x_1. \langle [(x z)] \mid x_1.\alpha_1 \rangle \rangle \\ &\Rightarrow \lambda x.\lambda y.\lambda z.\mu\alpha_1. \langle \mu\alpha_2. \langle z \mid \tilde{\mu}x_2. \langle y \mid x_2.\alpha_2 \rangle \rangle \mid \tilde{\mu}x_1. \langle \mu\alpha_3. \langle z \mid \tilde{\mu}x_3. \langle x \mid x_3.\alpha_3 \rangle \rangle \mid x_1.\alpha_1 \rangle \rangle \end{aligned}$$

While I provide only the skeleton of the type derivation of the above term in this report, one may indeed verify using the typing rules of $\bar{\lambda}\mu\tilde{\mu}$ that the type of the above term has the principal type of the S . At the root of the derivation tree is the following sequent:

$$\begin{array}{l} \Gamma \vdash_{LK\mu\tilde{\mu}} S' : (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)) \mid \Delta \\ \text{where} \\ S' = \lambda x.\lambda y.\lambda z.\mu\alpha_1. \langle \mu\alpha_2. \langle z \mid \tilde{\mu}x_2. \langle y \mid x_2.\alpha_2 \rangle \rangle \mid \tilde{\mu}x_1. \langle \mu\alpha_3. \langle z \mid \tilde{\mu}x_3. \langle x \mid x_3.\alpha_3 \rangle \rangle \mid x_1.\alpha_1 \rangle \rangle \end{array}$$

The derivation tree is as follows. Only the major connective that is relevant at each rule is shown so as to reduce the syntactic clutter. The tree is derived via structural induction on the term syntax.

$$\begin{array}{c}
\frac{\bar{z} \text{ AxR} \quad \frac{\bar{y} \text{ AxR} \quad \frac{\bar{x}_2 \text{ AxR} \quad \bar{\alpha}_2 \text{ AxL}}{x_2.\alpha_2} \rightarrow L \quad \text{Cut}}{\langle | \rangle} \text{ ActivateL}}{\langle | \rangle} \text{ ActivateR} \quad \frac{\bar{z} \text{ AxR} \quad \frac{\bar{x} \text{ AxR} \quad \frac{\bar{x}_3 \text{ AxR} \quad \bar{\alpha}_3 \text{ AxL}}{x_3.\alpha_3} \rightarrow L \quad \text{Cut}}{\langle | \rangle} \text{ ActivateL}}{\langle | \rangle} \text{ ActivateR} \quad \frac{\bar{x}_1 \text{ AxR} \quad \bar{\alpha}_1 \text{ AxL}}{x_1.\alpha_1} \rightarrow L \quad \text{Cut}}{\langle | \rangle} \text{ ActivateL} \\
\hline
\frac{\langle | \rangle}{\mu\alpha_2} \text{ ActivateR} \quad \frac{\langle | \rangle}{\mu\alpha_3} \text{ ActivateR} \quad \frac{\langle | \rangle}{\mu x_1} \text{ ActivateL} \\
\hline
\frac{\langle | \rangle}{\mu\alpha_1} \text{ ActivateR} \\
\frac{\lambda z}{\lambda y} \rightarrow R \\
\frac{\lambda y}{\lambda x} \rightarrow R
\end{array}$$

The the above derivation tree constitutes a proof of the principal type of S in the $LK\mu\tilde{\mu}$ calculus. The general approach shown here can be used to derive the proof on any proposition that is the principle type of a lambda term.

Part II

Control Operations

The principal type system of the simply typed lambda calculus corresponds to the implicational fragment of intuitionistic logic. The addition of control operators extends the computational calculus such that the principal type scheme corresponds to various other logics.

6 CallCC

The discussion on control operators starts with the callcc operator. The callcc operator is available as a primitive in the Scheme programming language. Callcc along with assignment or unbounded recursion (via letrec) enable Scheme programs to express all forms of computational effects. The operational semantics of callcc are specified by the abstract machine below.

6.1 Abstract Machine for CallCC

The following is the term language in which we define callcc. It is essentially an extension of the CBV lambda calculus abstract machine specified earlier.

Syntactic Categories:

$$\begin{aligned}
 \text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \mid \text{callcc } e \mid \mathcal{A} e \\
 \text{Syntactic Values, } v &= \lambda x.e \\
 \text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\
 \\
 \text{Evaluation Contexts, } E &= \square \mid v E \mid E e
 \end{aligned}$$

Reductions:

$$\begin{aligned}
 E[(\lambda x.e) v] &\longmapsto E[e[v/x]] \\
 E[\text{callcc } e] &\longmapsto E[e (\lambda x.A E[x])] \\
 E[\mathcal{A} e] &\longmapsto e
 \end{aligned}$$

6.2 Principal Type Scheme of CallCC: Pierce's law

The type of callcc as specified in the Haskell's Cont monad is:

$$\text{Control.Monad.Cont.callCC} :: \text{MonadCont } m \Rightarrow ((a \rightarrow m b) \rightarrow m a) \rightarrow m a$$

If we strip away the monadic decoration of effects from the type, we get the following:

$$\text{callCC} :: ((a \rightarrow b) \rightarrow a) \rightarrow a$$

This is essentially Pierce's law. Hence adding callcc as a primitive causes the type scheme to shift from implicational intuitionistic logic to what is sometimes referred to as 'minimal classical logic'. It is classical in the sense that adding callcc to the language enables multiple values to be returned to the same context.

Minimal Classical Logic:

$$\begin{aligned}
 S &:: (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)) \\
 K &:: a \rightarrow (b \rightarrow a) \\
 \text{callCC} &:: ((a \rightarrow b) \rightarrow a) \rightarrow a \\
 \text{Modus Ponens} &\Rightarrow a, a \rightarrow b \vdash b
 \end{aligned}$$

6.3 μ is similar to callcc

At this point it is useful to make the following observation about the μ operator of the $\bar{\lambda}\mu\tilde{\mu}$ calculus. The semantics of μ is reminiscent of callcc in the sense that it grabs the current context makes it available to the abstracted expression.

7 Other Control Operators

7.1 Abort: *ex falso quodlibet*

The ‘abort’ operator was represented as \mathcal{A} in the abstract machine for callcc. Abort essentially discards its current context. The principal type of abort is:

$$\mathcal{A} :: \perp \rightarrow a$$

Adding abort to the implicational fragment of intuitionistic logic gives us the full intuitionistic logic. Intuitionistic Logic:

$$\begin{aligned} S &:: (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)) \\ K &:: a \rightarrow (b \rightarrow a) \\ \mathcal{A} &:: \perp \rightarrow a \\ \text{Modus Ponens} &\Rightarrow a, a \rightarrow b \vdash b \end{aligned}$$

7.2 Felleisen’s C operator: double negation

Felleisen’s C [3] operator differs from callcc in that it escapes its current context. Here is the operational semantics:

$$\begin{aligned} \text{Expressions, } e &= x \mid \lambda x.e \mid e_1 e_2 \mid \mathcal{C} e \mid \mathcal{A} e \\ \text{Syntactic Values, } v &= \lambda x.e \\ \text{Denumerable Term Variables} &= x_1 \mid x_2 \dots \\ \text{Evaluation Contexts, } E &= \square \mid v E \mid E e \end{aligned}$$

Reductions:

$$\begin{aligned} E[(\lambda x.e) v] &\mapsto E[e[v/x]] \\ E[\mathcal{C} e] &\mapsto [e (\lambda x.A E[x])] \\ E[\mathcal{A} e] &\mapsto e \end{aligned}$$

The principal type of C is:

$$\mathcal{C} :: ((a \rightarrow \perp) \rightarrow \perp) \rightarrow a$$

The principal type of C is essential the double negation rule. Hence adding C as a constant to the simply type lambda calculus corresponds to having classical logic.

$$\begin{aligned} S &:: (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c)) \\ K &:: a \rightarrow (b \rightarrow a) \\ \mathcal{C} &:: ((a \rightarrow \perp) \rightarrow \perp) \rightarrow a \\ \text{Modus Ponens} &\Rightarrow a, a \rightarrow b \vdash b \end{aligned}$$

7.3 CPS erases Control operations

CPS converting a term language with control operations erases the control operations. I provide here a CBV CPS translation that shows the erasure of abort and callcc. There exist similar CPS translations for C and other control operations.

$$\begin{aligned} [x] &= \lambda k.(k x) \\ [\lambda x.e] &= \lambda k.(k \lambda x.[e]) \\ [e_1 e_2] &= \lambda k.([e_1] \lambda f.([e_2] \lambda v.(f v k))) \\ [\mathcal{A} e] &= \lambda k.([e] \lambda x.x) \\ [\text{callcc } e] &= \lambda k.([e] \lambda f.(f \lambda x.(\lambda u.(k x)) k)) \end{aligned}$$

The translation of callcc shows how the continuation passed to f is an abortive one since it ignores its current continuation bound to the variable u .

8 Control Operators and Logics they induce

Addition of control operators, makes the principal type scheme of the language correspond to various logics. Adding various control operators to ones language enables you to have expressive power corresponding to various logics. One can think of this ‘expressive power’ as the ability to write programs which are proofs of propositions in various logics.

The lambda calculus extended with the C operator is computationally equivalent to the Parigot’s $\lambda\mu$ calculus [8]. The $\lambda\mu$ calculus corresponds natural deduction while $\bar{\lambda}\mu$ calculus (a subset of $\bar{\lambda}\mu\tilde{\mu}$ without the $\tilde{\mu}$ [6] operator) corresponds to natural deduction. The translation from from lambda calculus to $\bar{\lambda}\mu\tilde{\mu}$ that was presented earlier is essentially the subset of the translation of $\lambda\mu$ to $\bar{\lambda}\mu\tilde{\mu}$ by omitting the μ terms.

The discussion of control operators in relevant here to point the following fact: While the addition of control operators changes the axioms available in the logic corresponding to the principle type scheme, they do not influence the proof theoretic approach. In other words addition of all of the above control operators essentially keeps the type derivations in natural deduction. The shift occurs from natural deduction to sequent calculus not because of the expressive power of the language with respect to control operations, but because of a fundamental change in the way environments are treated.

While the μ operator of the $\bar{\lambda}\mu\tilde{\mu}$ calculus is similar to C the $\tilde{\mu}$ operator is very different. The $\tilde{\mu}$ operator essentially ‘gets the value of a variable’. When $\tilde{\mu}$ appears in a right hand side of a ‘command’ it grabs the value of the term variable ‘x’. This is in some ways reminiscent of the environment lookup operation $\rho(x)$ that is the used in the CEK style abstract machines presented for the CBV/CBN lambda calculus. This $\tilde{\mu}$ operation essentially is a shift in the way binding environments are represented which is the essential shift required from moving from natural deduction to sequent calculus.

Adding control operators to the language does not directly give as a different encoding of the environments. Which maybe an intuitive explanation why they do not affect the proof theoretic approach required for typing the language.

The call-by-need lambda calculus however forces one to look at the binding environments for variables in a completely different way. Hence it is tempting to examine call-by-need in the context of the environment manipulating operations. To make all of this more precise I go on to present a formalization of a set of control operations referred to as delimited continuations and use delimited continuations to encode and manipulate environments of the runtime stack. This I believe is a useful step into formalizing a calculus and CPS translation that treats call-by-needs as a sequence of control and environment manipulation operations.

9 Delimited Continuations

I use the formalization of delimited continuations as proposed by Kent Dybvig, Simon Peyton Jones and Amr Sabry [7]. This formalisation consists of four basic operations. In their paper they show an encoding of all other control operations such as callcc , C , F operators, shift-reset etc using these four operations. The formalization provided by used a monadic encapsulation of effects and hence the types provided below execute in the context of a delimited continuation monad called the CC monad.

Monad CC

$$\begin{aligned} \text{newP} &:: CC \text{ (Prompt } a) \\ \text{pushP} &:: \text{Prompt } a \rightarrow CC \ a \rightarrow CC \ a \\ \text{withSC} &:: \text{Prompt } a \rightarrow (\text{SubCont } b \ a \rightarrow CC \ a) \rightarrow CC \ b \\ \text{pushSC} &:: \text{SubCont } a \ b \rightarrow CC \ a \rightarrow CC \ b \end{aligned}$$

While explaining the details of delimited continuations is beyond the scope of this report, here is the intuition behind the operators. The newP operator creates a fresh ‘prompt’. A prompt can be thought of as a delimiter for the runtime stack of a program. A prompt maybe pushed onto runtime stack using the pushP operator. The withSC operator is invoked with a prompt as one of its arguments. The withSC operator essentially grabs the continuation from the current top of the runtime stack to the point at which prompt is found on the stack. Such a continuation is sometimes referred to as a sub-continuation and hence the parametric type SubCont . The pushSC operator essentially pushes a captured continuation back onto the runtime stack.

These semantics are expressed in the reduction rules below:

$$\begin{aligned}
E[\text{new}P] &\mapsto E[p] \text{ where } p \text{ is a fresh prompt} \\
E[\text{push}P \ p \ E'[\text{with}SC \ p \ e]] &\mapsto E[e \ (\text{cont } E')] \\
E[\text{push}P \ p \ v] &\mapsto E[v] \\
E[\text{push}SC \ (\text{cont } E') \ e] &\mapsto E[E'[e]]
\end{aligned}$$

The above formalization of delimited continuations is used to encode call-by-need as a series of stack operations. In the translation provided later on. Similar to other control operations a CPS erasure of delimited continuations is known.

10 The Call-By-Need Lambda Calculus

The call-by-need lambda calculus sometimes known as ‘lazy evaluation’ falls somewhere in between CBV and CBN. CBNeed terminates exactly when CBN terminates and it does reduction steps comparable to reduction steps taken by CBV.

The essential idea is this: when a CBNeed evaluator encounters a redex of the form $(\lambda x.e_1)e_2$, instead of evaluating e_2 , the evaluator proceeds similar to CBN by binding x to e_2 and then evaluating the body of the lambda expression, namely e_1 . When x is encountered during the evaluation of e_1 the first time, the expression bound to x , namely e_2 , is evaluated. So far evaluation is similar to CBN evaluation. However at this point it differs from CBN in that once e_2 is evaluated to a value (say v), the environment of e_1 is permanently destructively updated such that x is bound to v . All subsequent references of x will return the value v . A few interesting points to note are that if x is never referenced in the evaluation of e_1 , then e_2 is never evaluated. Hence if e_2 was a non terminating computation and e_1 was not CBN and CBNeed evaluation would terminate while CBV evaluation will not. Similarly if x was referenced several times during the evaluation of CBNeed and CBV evaluation will evaluate e_2 only once while CBN evaluation will evaluate e_2 as many times as x is referenced.

Traditionally CBNeed is explained using environments that bind computations to variables and destructive update of the environment on variable lookup. However such a assignment operation is a side effect and is hence not conducive to equational reasoning. However in 1993 a pure abstract machine for CBNeed was proposed by Zena Ariola and Matthias Felleisen [?].

10.1 Abstract Machine by Zena Ariola et al

The existence of the above abstract machine [?] causes one to suspect that a CPS for the CBNeed calculus exists. However it is not obvious what such a CPS transformation is. Close inspection of the reduction rules reveals that the machine uses a different approach to binding environments. In some sense environments are encoded on the runtime stack. This is in many ways reminiscent of the $\bar{\lambda}\mu\tilde{\mu}$ calculus.

$$\begin{aligned}
\text{Expressions, } e &= x \mid \lambda x.e \mid e_1 \ e_2 \\
\text{Values, } v &= \lambda x.e \\
\text{Answers, } a &= v \mid ((\lambda x.a) \ e) \\
\text{Contexts, } E &= \square \mid E \ e \mid ((\lambda x.E) \ e) \mid ((\lambda x.E[x]) \ E)
\end{aligned}$$

The definitions above introduce the syntactic category of answers, a. Unlike the other definition of lazy evaluation the formalization below does not do a destructive update of an environment.

$$\begin{aligned}
E[(\lambda x.E[x]) \ v] &\mapsto E[(\lambda x.E[v]) \ v] \\
E[(\lambda x.E[x]) \ ((\lambda y.a) \ e)] &\mapsto E[(\lambda y.(\lambda x.E[x]) \ a) \ e] \\
E[(\lambda x.a) \ e_1 \ e_2] &\mapsto E[(\lambda x.a \ e_2) \ e_1]
\end{aligned}$$

11 Call-by-Need using delimited continuations

The following is the translation of a CBNeed term language into a CBV language. The following formulation of CBNeed encodes environments of the runtime stack of the programs using delimited control operations and unbounded recursion. The stack manipulations done by this translation are similar to the the reduction rules of the abstract machine presented above. The translation also shows support for constants and primitive operations.

$$\begin{aligned}
T[x] &= \text{yield } x \\
T[\lambda x.e] &= \lambda k.(k \lambda p.\text{pushNewP } x \text{ (captureCls } p \text{ } T[e])) \\
T[e_1 e_2] &= \text{pushNewP } p \text{ (} T[e_1] \text{ } \lambda f.(\text{let } k = (f \text{ } p), v = T[e_2] \text{ in applyCls } k \text{ } v)) \\
\\
T[n] &= \lambda k.(k \text{ } n) \\
T[e_1 * e_2] &= \text{app } ((* \text{ (removeCls } T[e_1])) \text{ (removeCls } T[e_2])) \\
\\
\text{where} \\
\text{captureCls } p \text{ } v &= \text{withSC } p \lambda k.\lambda f.(\text{pushSC } k \text{ (} v \text{ } f)) \\
\text{applyCls } k \text{ } v &= v \lambda x.(\text{rec } k \lambda k.(k \text{ } x)) \\
\text{removeCls } v &= \text{pushNewP } p \text{ (} v \lambda x.(\text{withSC } \lambda k.x)) \\
\\
\text{pushNewP } p \text{ } [e] &= \text{let } p = \text{newP in (pushP } p \text{ } [e]) \\
\text{yield } p &= \text{withSC } p \lambda k.\lambda x.(\text{pushP } p \text{ (pushSC } k \text{ } x)) \\
\\
\text{app} &= \lambda x.\lambda k.(k \text{ } x) \\
\text{rec } a \text{ } b &= \text{rec (} a \text{ } b)
\end{aligned}$$

Unbounded recursion is introduced as a primitive in the form of the combinator ‘rec’. A detailed description of the above translation is beyond the scope of this report. However it is useful to note that CPS translations exist for delimited control operations and hence one may derive a CPS for Call-by-need in this fashion. However the translation to control operators is rather unintuitive, its correctness has not been proved (though I have implemented an interpreter and verified the translation) and it is a rather cumbersome translation. These shortcomings essentially prevent us from considering this a good CPS for the CBNeed calculus.

A good translation would expose two categories of operators for stack manipulation and environment manipulation in much the same way that the $\lambda\mu\tilde{\mu}$ calculus does. Such a translation and a term language is a topic of research.

12 Conclusions

The investigation of a CPS for CBNeed has opened a pandora’s box of interconnections between various computational calculi and their relationships to proof theoretic formulations via the Curry-Howard Isomorphism. It would be interesting to discover a term language suitable for expressing the CBNeed calculus which is expressible as the combination of a pure calculus + ‘control’ operations + ‘environment’ operations. Such a calculus would might provide insights into the relationship between natural deduction and sequent calculi and provide a CPS for CBNeed.

References

- [1] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 1995.
- [2] Grard Boudol, Inria Sophia Antipolis, and Sophia Antipolis Cedex. On the semantics of the call-by-name cps transform. 1999.
- [3] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM SIGPLAN Notices*, 35(9):233–243, 2000.

- [4] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52(3):205–237, 1987.
- [5] G. Gentzen. Investigations into logical deduction. In *M. E. Szabo, editor, The Collected Papers of Gerhard Gentzen, North-Holland, Amsterdam*, 1969.
- [6] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Conf. Record 21st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Oregon, PL, USA, 17–21 Jan. 1994*, pages 458–471. ACM Press, New York, 1994.
- [7] Hugo Herbelin. Investigations into the duality of computation.
- [8] Simon Peyton Jones Kent Dybvig and Amr Sabry. A monadic framework for delimited continuations. 2005.
- [9] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 190–201, London, UK, 1992. Springer-Verlag.
- [10] Gordon Plotkin. Call-by-name, call-by value and the lambda calculus. 1975.
- [11] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, 1965.
- [12] Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 189–201, New York, NY, USA, 2003. ACM Press.
- [13] Aaron Bohannon Zena M. Ariola and Amr Sabry. Sequent calculi and abstract machines. 2006.