Sequent Combinators: A Hilbert System for the Lambda Calculus

Healfdene Goguen
Department of Computer Science, University of Edinburgh
The King's Buildings, Edinburgh, EH9 3JZ, United Kingdom
Fax: (+44) (131) 667-7209

Abstract

This paper introduces a Hilbert system for lambda calculus called sequent combinators. Sequent combinators address many of the problems of Hilbert systems, which have led to the more widespread adoption of natural deduction systems in computer science. This suggests that Hilbert systems, with their more uniform approach to meta-variables and substitution, may be a more suitable framework than lambda calculus for type theories and programming languages.

1 Introduction

Hypothetical reasoning is an important tool in logic. However, although most logics include mechanisms for hypothetical reasoning, there is an important divide in how this reasoning is admitted:

- By introducing basic principles of reasoning and then proving in the metatheory that the deduction theorem holds, saying that if B is provable under hypothesis A then $A \Rightarrow B$ is provable. We shall call such systems *Hilbert systems*.
- By including the deduction theorem as a rule of inference, making hypothetical reasoning a first-class construct. We shall call such systems *natural deduction systems*, because Gentzen first internalized the deduction theorem in this way in his system of natural deduction [9].

This distinction lifts smoothly to type theory with first-class proof objects. Curry's combinators with application [6] encode Hilbert's axioms with the rule of inference modus ponens, and the abstraction algorithm captures the deduction theorem. Church's typed lambda calculus [2] gives a language of proofs for Gentzen's natural deduction.

Hilbert systems have some important advantages over natural deduction systems. The metatheoretic notion of substitution is simple textual substitution in Hilbert systems. This

means that Hilbert systems are easier to understand intuitively, to implement, and to reason about. Indeed, early papers [3] seem to have justified the internalization of the deduction theorem as lambda abstraction on the basis of Hilbert systems and the deduction theorem.

However, natural deduction systems have come to dominate computer science for several reasons:

- The size of derivations or proof objects explodes using the deduction theorem in Hilbert systems.
- Meta-properties, such as the deduction theorem, are commonly brought into the object language. Systems of explicit substitution [1] further internalize substitution and weakening.
- Lambda calculus reduction is stronger and more natural than combinator reduction. The ξ-equality of lambda calculus, which allows equality under lambda abstraction or in hypothetical proofs, cannot be captured naturally by reduction on combinators.
- The duality between introduction and elimination for the logical constants in natural deduction systems or sequent calculus doesn't appear in Hilbert systems.

This paper aims to demonstrate that these defects of Hilbert systems can be overcome by introducing a new Hilbert system, sequent combinators. Much of the technical work in the paper will be devoted to the third point, showing that the reduction for lambda calculus is preserved by the translation to sequent combinators. The other points will be dealt with in the course of the paper. Together with the benefits of a simpler metatheory and easier implementations, this provides a basis for believing that Hilbert systems could be used to underlie type-theory based proof assistants and even functional programming language implementations.

In general we shall omit proofs in this paper, because they are similar to well-known proofs of similar properties for combinators [6].

1.1 Sequent Combinators

It is not commonly known that Gentzen gave two styles of interpretation of sequent calculus, interpreting intuitionistic sequents $A_1, \ldots, A_m \vdash B$ either in what could be called Hilbert style as the formula $A_1 \wedge \cdots \wedge A_m \Rightarrow B$, or in what could be called natural deduction style as B under assumptions $A_1 \ldots A_m$. We take inspiration from the Hilbert style interpretation to define a new language of combinators, formulating a calculus with implicit currying where we read the above sequent as $A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow B$. The language has term constructors corresponding to rules of inference similar to those of sequent calculus, rather than constants or combinators corresponding to axioms in Hilbert's formal systems. We shall show that the new language is sound and complete for the lambda calculus, including the equalities α, β, η and ξ.

The intuition underlying Curry's combinators is that they represent particular λ-terms, or functions, under α-equivalence. The basic operations of our sequent combinators are just Curry's, namely the identity I, composition S, and projection K. However, our combinators are instead indexed by a fixed number of arguments. Hence, our combinator $\mathrm{K}_m(M)$, corresponding to K, takes the first m arguments and passes them on to M, but ignores the $m+1$'th argument. This is in contrast to K's simpler behavior of taking two arguments and returning the first. Our combinators for I and S are also parameterized by a number of arguments. The completeness proof in Section 4 gives a more formal intuition for the meaning of the combinators, by giving lambda terms corresponding to each combinator.

Alternatively, we notice that the sequent combinators we introduce corresponding to I, K and S are similar to the rules of inference for variables, thinning and substitution for lambda calculus. We pursue this similarity further in Section 7.

As we have stated above, sequent combinators have important metatheoretic properties. In comparison with combinators, our equalities can naturally be interpreted as reductions, and the full reduction of lambda calculus is preserved by translation into sequent combinators. In comparison with systems of explicit substitution, the equality β is satisfied with respect to the metatheoretic, textual substitution.

1.2 Related Work

The goals of our work are similar to those of Curry, but there are important differences in the two approaches. Curry models the lambda calculus using a basic set of combinators, namely S and K, together with a collection of equations to explain their behavior. However, S and K are so basic that complex equations are needed to capture the rules η (expressing a weak extensionality of functions in the lambda calculus) and especially ξ (expressing the compatibility of the constructor λ with equality). Although model-theoretically these equations are quite easy to verify, they are difficult to understand in the modern treatment of type theory and term rewriting, where we treat equalities as reduction rules from left to right.

This is an improvement of our work over Curry's. The equalities over our combinators can also easily be understood as reduction rules by interpreting them as rewriting from left to right. Our combinators extend combinatory logic in a natural way. The combinators S and K, as well as the basic operation of application, are captured as particular instances of our combinators. Soundness of our system for Curry's equations follows straightforwardly because both our calculus and his are sound and complete for lambda calculus.

Furthermore, reduction in lambda calculus is preserved by the translation into our sequent combinators. Hence, questions of strong normalization of typed lambda calculi can be studied in the context of a bound variable-free language. We therefore introduce typing rules for both languages.

A disadvantage of our representation is that we introduce families of combinators indexed by natural numbers. This is more complicated than the simple language with only two or three simple combinators.

Combinators have been used in implementations of functional programming following Turner [20], leading to such notions as supercombinators [13], lambda-lifting [14], and director strings [16]. However, these implementations do not consider the problem of equality of expressions involving the abstraction operator, and in particular the ξ-rule is not sound.

Systems of explicit substitution (Abadi et al. [1], among many others) and categorical combinators [5] put substitution in the object language rather than leaving it as a metatheoretic notion. In our formulation, the metatheoretic substitution is simple textual substitution, rather than the complex substitution of lambda calculus (see Section 2). Furthermore, while our system satisfies equalities such as β and ξ with respect to the metatheoretic notions of abstraction and substitution, the actual reductions of the system do not use the metatheoretic definitions. We shall discuss the relationship of our work with the system λs of Kamareddine and Ríos [15] in Section 7.

Categorical logic, in particular as developed by Lambek and Scott [17], has several of the features we mention in the introduction. In particular, substitution is internalized, and the relationship between the object and meta-substitutions is clear. Furthermore, logical constants are presented in the usual manner of category theory as limit diagrams.

However, there are problems with categorical logic from the point of view of the intensional type theory used in modern proof assistants. First, the reduction of classical lambda calculus is not captured by reduction in categorical logic. Secondly, the size of derivations seems to increase significantly under the abstraction algorithm. Finally, and most importantly, the extensionality inherent to category theory is too strong for more sophisticated type theories. It is a folk result in type theory that extensionality with absurdity and type universes leads to non-termination and undecidability of type checking (see for example [12]). Categorical logic relies on the extensionality to capture equality of lambda calculus.

Martin-Löf [18] investigated dependent type theory with the restricted reduction which does not reduce under binders. However, he later abandoned this system in favor of stronger systems with reduction under binders, apparently because the earlier systems were too weak.

1.3 Overview

The paper is organized as follows. Section 2 introduces the typing rules for simply-typed lambda calculus. Section 3 introduces our combinators and typing rules for them. Section 4 proves completeness of the combinators for lambda calculus. Section 5 proves soundness of the combinators for lambda calculus, using functional abstraction. Section 6 discusses the logical connectives and inductive types, demonstrating that the symmetry between introduction and elimination of natural deduction can be preserved in Hilbert systems. Section 7 introduces abstraction as a constructor of the language and considers the similarity of the resulting system with de Bruijn terms with explicit substitution. Section 8 discusses further work and Section 9 draws conclusions.

2 The Simply-Typed Lambda Calculus

We assume the existence of a denumerable set V of variables $x, y, z, \ldots$

The types, pre-contexts and terms are introduced by the grammar:

$$\begin{array}{rcl} A, B, C \in Ty & ::= & o \mid A \Rightarrow B \\ \Gamma, \Delta, \Phi \in C & ::= & () \mid \Gamma, x{:}A \\ M, N, P \in T & ::= & x \mid M(N) \mid \lambda x.M \end{array}$$

A pre-context $x_1{:}A_1, \ldots, x_m{:}A_m$ is a context if the x_i are distinct.

We give the definition of substitution in full, because this paper studies properties of substitution and bound variables in depth.

$$\begin{array}{rcll} [N/x]x & =_{\mathrm{df}} & N & \\ [N/x]y & =_{\mathrm{df}} & y & \\ [P/x]M(N) & =_{\mathrm{df}} & ([P/x]M)([P/x]N) & \\ [N/x](\lambda x.M) & =_{\mathrm{df}} & \lambda x.M & \\ [N/x](\lambda y.M) & =_{\mathrm{df}} & \lambda y.[N/x]M & (x \neq y \textit{ and } y \notin \mathrm{FV}(N)) \\ [N/x](\lambda y.M) & =_{\mathrm{df}} & \lambda z.[N/x][z/y]M & (z \notin \mathrm{FV}(MN) \textit{ and } y \in \mathrm{FV}(N)) \end{array}$$

We write R^c for the compatible closure of a relation R. This includes the rule ξ, which says that if MR^cN then $(\lambda x.M)R^c(\lambda x.N)$. We say that two terms M and N are α-equivalent, $M \equiv N$, if they are related by the compatible closure of the following rule:

$$\lambda x.M \quad \alpha \quad \lambda y.[y/x]M \quad (y \notin \mathrm{FV}(M))$$

The judgement form for this calculus is as usual, $\Gamma \vdash M : A$. The rules of inference for the calculus are as follows:

$$(\textit{Var}) \quad \frac{\Gamma \text{ context} \quad x{:}A \notin \Gamma}{\Gamma, x{:}A \vdash x : A}$$

$$(\textit{Thin}) \quad \frac{\Gamma_0, \Gamma_1 \vdash M : A \quad z \notin \mathit{dom}(\Gamma_0, \Gamma_1)}{\Gamma_0, z{:}C, \Gamma_1 \vdash M : A}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

$$(\textit{App}) \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B}$$

$$(\textit{Subst}) \quad \frac{\Gamma_0, z{:}C, \Gamma_1 \vdash M : A \quad \Gamma_0 \vdash N : C}{\Gamma_0, \Gamma_1 \vdash [N/z]M : A}$$

$$(\alpha) \quad \frac{\Gamma \vdash M : A \quad M \equiv N}{\Gamma \vdash N : A}$$

We also introduce the usual notions of $\beta\eta$-equality and reduction on terms in the lambda calculus:

$$\begin{array}{rcll} (\lambda x.M)(N) & \beta & [N/x]M & \\ \lambda x.(M(x)) & \eta & M & (x \notin \mathrm{FV}(M)) \end{array}$$

Reduction $M \rhd N$ is the compatible closure of these rules; $M \rhd^+ N$ is the transitive closure of $\rhd$; $M \rhd^* N$ is the reflexive, transitive closure of $\rhd$; and $M = N$ is the least equivalence relation containing $\rhd$.

3 A Sequent Calculus with Combinator Proof Terms

In this section we introduce the combinatory language informally related to the sequent calculus. It is informally related because the expression of the rules of inference is similar to sequent calculus, but the constants underlying the rules more closely resemble combinatory logic.

3.1 Terms and Rules of Inference

Let $m, n, p, \ldots$ be arbitrary natural numbers. Types and contexts in this calculus are as in the simply typed lambda calculus. The notion of a variable-free context, simply a list of types, and the terms of this language are defined by the grammar:

$$\begin{array}{rcl} X, Y, Z \in L & ::= & () \mid X, A \\ M, N, P \in T & ::= & x \mid \mathrm{I}_m \mid \mathrm{K}_m(M) \mid \mathrm{S}_m(M, N) \end{array}$$

If X is a variable-free context, then its length $|X|$ is simply its length as a list.

The judgement form for this calculus is $\vdash_\Gamma M : A$. The informal meaning of this judgement is that it encodes a variable-free language for proofs in the simply typed lambda calculus. The important difference is that the rules of inference are expressed with respect to the variable-free contexts.

We would interpret the usual judgement $X \vdash A$ in sequent calculus, where the proof contains no free variables, by saying $\vdash_{()} M : X \to A$ for some term M, where $X \to A$ is defined as:

- $() \to A =_{\mathrm{df}} A$.
- $(X, B) \to A =_{\mathrm{df}} X \to (B \Rightarrow A)$.

We now give the rules of inference. We understand in each rule that Γ is a context. We write the rules in a way which is suggestive of the sequent calculus, where $\to$ informally

plays the role of $\vdash$ in sequent calculus:

$$(Par) \qquad \frac{x{:}A \in \Gamma}{\vdash_\Gamma x : \ () \to A}$$

$$(\mathrm{I}) \qquad \frac{}{\vdash_\Gamma \mathrm{I}_{|X|} : \ (X, A) \to A}$$

$$(\mathrm{K}) \qquad \frac{\vdash_\Gamma M : \ X \to A}{\vdash_\Gamma \mathrm{K}_{|X|}(M) : \ (X, B) \to A}$$

$$(\mathrm{S}) \qquad \frac{\vdash_\Gamma M : \ (X, A) \to B \quad \vdash_\Gamma N : \ X \to A}{\vdash_\Gamma \mathrm{S}_{|X|}(M, N) : \ X \to B}$$

We can see how the usual combinators are represented in this calculus:

- Application $M(N)$ is simply the sequent combinator $\mathrm{S}_0(M, N)$.
- K corresponds informally to the sequent combinator $\mathrm{K}_1(\mathrm{I}_0)$ (this follows easily by using the translation of Definition 5.2 on the term $\lambda x.\lambda y.x$).
- S corresponds informally to the sequent combinator $\mathrm{S}_3(\mathrm{K}_1(\mathrm{I}_0), \mathrm{I}_1)$ (this again follows by using the translation on the term $\lambda x.\lambda y.\mathrm{S}_1(x, y)$, or on the term $\lambda x.\lambda y.\lambda z.(xz)(yz)$ and normalizing).

3.2 Equality and Reduction

We have the substitution equalities:

$$\begin{array}{rcl}
\mathrm{S}_m(\mathrm{I}_m, M) & = & M \\
\mathrm{S}_m(\mathrm{I}_{m+1+n}, M) & = & \mathrm{I}_{m+n} \\
\mathrm{S}_m(\mathrm{K}_m(M), N) & = & M \\
\mathrm{S}_m(\mathrm{K}_{m+1+n}(M), N) & = & \mathrm{K}_{m+n}(\mathrm{S}_m(M, N)) \\
\mathrm{S}_m(\mathrm{S}_{m+1+n}(M, N), P) & = & \mathrm{S}_{m+n}(\mathrm{S}_m(M, P), \mathrm{S}_m(N, P))
\end{array}$$

and the thinning equalities:

$$\begin{array}{rcl}
\mathrm{K}_m(\mathrm{I}_{m+n}) & = & \mathrm{I}_{m+1+n} \\
\mathrm{K}_m(\mathrm{K}_{m+n}(M)) & = & \mathrm{K}_{m+1+n}(\mathrm{K}_m(M)) \\
\mathrm{K}_m(\mathrm{S}_{m+n}(M, N)) & = & \mathrm{S}_{m+1+n}(\mathrm{K}_m(M), \mathrm{K}_m(N))
\end{array}$$

as well as the usual compatible closure.

Similar to the lambda calculus, we have both syntactic equality $M \equiv N$ and provable equality $M = N$. However, in the combinatorial language syntactic equivalence is exactly syntactic identity, rather than α-equivalence in the lambda calculus. Reduction is again defined by reading the equality rules from left to right.

The standard results about reduction hold for this calculus.

Proposition 3.1 (Church-Rosser) *The reduction relation is locally confluent.*

Proposition 3.2 (Subject Reduction) *If* $\vdash_\Gamma M : A$ *and* $M \rhd N$ *then* $\vdash_\Gamma N : A$.

Weaker versions of the thinning equalities have appeared in the functional programming literature [19], for example the equality $S(K(M), K(N)) = K(M(N))$.

3.3 η-Equality

In order to define the η-equalities we need the following definition, corresponding to a variable being free in a term in classical lambda calculus.

Definition 3.3 *A natural number* m *is* free *in* M *if one of the following clauses holds:*

- m *is free in* $\mathrm{K}_m(M)$.
- m *is free in* I_{m+1+n}.
- m *is free in* $\mathrm{K}_{m+1+n}(M)$ *if* m *is free in* M.
- m *is free in* $\mathrm{S}_{m+1+n}(M, N)$ *if* m *is free in* M *and* m *is free in* N.

The η-equalities can now be expressed as:

$$\begin{array}{rcl} \mathrm{S}_{m+1}(\mathrm{K}_m(M), \mathrm{I}_m) & = & M \\ \mathrm{S}_{m+1}(\mathrm{S}_{m+1}(M, N), \mathrm{I}_m) & = & \mathrm{S}_m(\mathrm{S}_{m+1}(M, \mathrm{I}_m), \mathrm{S}_{m+1}(N, \mathrm{I}_m)) \end{array}$$

both subject to the condition that m is free in M and N.

4 Completeness

We prove completeness of the sequent calculus for the lambda calculus first, to give a motivation for the combinators we have introduced.

We introduce the notation $\lambda^m \bar{x}.M$, where $\bar{x}$ is a list of variables $x_1 \ldots x_m$, as $\lambda x_1. \ldots \lambda x_m.M$. Similarly, $M^m(\bar{x})$ with $\bar{x}$ a list of variables is simply $M(x_1) \ldots (x_m)$.

Definition 4.1 *We define the interpretation of terms of this calculus as follows:*

$$\begin{array}{rcl} [\![x]\!] & =_{\mathrm{df}} & x \\ [\![\mathrm{I}_m]\!] & =_{\mathrm{df}} & \lambda^m \bar{x}.\lambda a.a \\ [\![\mathrm{K}_m(M)]\!] & =_{\mathrm{df}} & \lambda^m \bar{x}.\lambda a.[\![M]\!]^m(\bar{x}) \\ [\![\mathrm{S}_m(M, N)]\!] & =_{\mathrm{df}} & \lambda^m \bar{x}.([\![M]\!]^m(\bar{x}))([\![N]\!]^m(\bar{x})) \end{array}$$

The interpretation has the important properties of preserving equality and typing.

Proposition 4.2 (Completeness for Equality) *If $M = N$ then $[\![M]\!] = [\![N]\!]$.*

Proposition 4.3 (Completeness for Typing) *If $\vdash_\Gamma M : X \to A$ then $\Gamma \vdash [\![M]\!] : X \to A$.*

We have introduced both a lambda calculus and a combinatory language with untyped terms. We could also present these systems with typed terms: the typed lambda calculus would be as it is usually presented with labels on abstraction, and the combinatory language would include full type information (so for example we would write $\mathrm{S}_m(M, N)$ in the inference rule S as $\mathrm{S}^A_X(M, N)$).

5 Soundness

Soundness extends Curry's programme of using combinators as a (bound) variable-free language for the lambda calculus. There are several steps to this:

- Functional Abstraction. This involves defining the abstraction operation $[x]M$ by induction on the structure of M (Definition 5.1).
- Textual Substitution. This involves defining the operation of textual substitution $M[x \leftarrow N]$ (Definition 5.3).
- Functional Completeness. This involves showing that the abstraction and substitution operations as defined are coherent, in the sense that the β-rule is satisfied (Lemma 5.6).
- Soundness. We extend Curry's results by also showing that the reduction and equality of lambda calculus is preserved by the translation to sequent combinators (Corollary 5.9 and Theorem 5.10).

The abstraction algorithm corresponds to the deduction theorem for intuitionistic implicational logic: if from $\Gamma, x{:}A$ we can prove B then we can prove $A \Rightarrow B$ from Γ. Technically, we can see that because the combinators include information about their list of arguments, the algorithm is particularly easy to define.

Definition 5.1 (Abstraction) *We define abstraction $[x]M$ in the combinator calculus by induction on M:*

$$\begin{array}{rcl}
[x]x & =_{\mathrm{df}} & \mathrm{I}_0 \\
[x]y & =_{\mathrm{df}} & \mathrm{K}_0(y) \\
[x]\mathrm{I}_m & =_{\mathrm{df}} & \mathrm{I}_{1+m} \\
[x]\mathrm{K}_m(M) & =_{\mathrm{df}} & \mathrm{K}_{1+m}([x]M) \\
[x]\mathrm{S}_m(M, N) & =_{\mathrm{df}} & \mathrm{S}_{1+m}([x]M, [x]N)
\end{array}$$

Notice that, unlike some definitions of abstraction in combinatory logic, we do not have a special case in the algorithm to define abstraction $[x]M$ for terms M such that $x \notin \mathrm{FV}(M)$ as $\mathrm{K}_0(M)$. This is because our calculus is sufficiently expressive that the reduction $\mathrm{K}_0(M) \rhd^* [x]M$ holds, using the thinning equalities.

We can now define a translation from the lambda calculus into the combinatorial language.

Definition 5.2 *Define the map M^* inductively on the structure of terms in the lambda calculus:*

$$\begin{array}{rcl} x^* & =_{\mathrm{df}} & x \\ (M(N))^* & =_{\mathrm{df}} & \mathrm{S}_0(M^*, N^*) \\ (\lambda x.M)^* & =_{\mathrm{df}} & [x](M^*) \end{array}$$

Definition 5.3 (Textual Substitution) *We define textual substitution $M[x \leftarrow N]$ in the combinator calculus by induction on M:*

$$\begin{array}{rcl} x[x \leftarrow M] & =_{\mathrm{df}} & M \\ y[x \leftarrow M] & =_{\mathrm{df}} & y \\ \mathrm{I}_m[x \leftarrow M] & =_{\mathrm{df}} & \mathrm{I}_m \\ \mathrm{K}_m(M)[x \leftarrow P] & =_{\mathrm{df}} & \mathrm{K}_m(M[x \leftarrow P]) \\ \mathrm{S}_m(M,N)[x \leftarrow P] & =_{\mathrm{df}} & \mathrm{S}_m(M[x \leftarrow P], N[x \leftarrow P]) \end{array}$$

Lemma 5.4 (α-Equivalence) *We have $[x]M \equiv [y](M[x \leftarrow y])$ if $y \notin \mathrm{FV}(M)$.*

Lemma 5.5 *The reduction $([x]M)[y \leftarrow N] \rhd^* [x](M[y \leftarrow N])$ holds if $y \notin \mathrm{FV}(N)$.*

Lemma 5.6 (Functional Completeness) *The equality β holds for the defined abstraction and textual substitution:*

$$\mathrm{S}_0([x]M, N) \quad \rhd^+ \quad M[x \leftarrow N]$$

The following lemma fails for ordinary combinators.

Lemma 5.7 (ξ-Reduction) *If $M \rhd N$ then $[x]M \rhd^+ [x]N$.*

Proof By induction on M such that $M \rhd N$.

Lemma 5.8 *We have $M^*[x \leftarrow N^*] \rhd^* (M[x \leftarrow N])^*$.*

Corollary 5.9 (Soundness for Reduction) *If $M \rhd N$ then $M^* \rhd^+ N^*$.*

Reduction in the simply-typed lambda calculus is preserved by reduction in the combinatory system. Hence we have a preservation of strong normalization result: if the combinatory calculus is strongly normalizing then so is the usual representation.

Theorem 5.10 (Soundness for Equality) *If $M = N$ then $M^* = N^*$.*

Lemma 5.11 *If $\vdash_{\Gamma_0, x:B, \Gamma_1} M : A$ then $\vdash_{\Gamma_0, \Gamma_1} [x]M : B \Rightarrow A$.*

Corollary 5.12 *If $\Gamma \vdash M : A$ then $\vdash_\Gamma M^* : A$.*

Proof By induction on derivations, using Subject Reduction together with Lemma 5.8 for (*Subst*), Lemma 5.11 for (λ), and Lemma 5.4 for (α).

5.1 Soundness for η

The treatment of the η reduction rules requires some auxiliary lemmas.

Lemma 5.13 *If m is free in M and $M \rhd N$ then m is free in N.*

Definition 5.14 *If m is free in M then define M^m as follows:*

$$\begin{array}{rcl} \mathrm{K}_m(M)^m & =_{\mathrm{df}} & M \\ \mathrm{I}_{m+1+n}{}^m & =_{\mathrm{df}} & \mathrm{I}_{m+n} \\ \mathrm{K}_{m+1+n}(M)^m & =_{\mathrm{df}} & \mathrm{K}_{m+n}(M^m) \\ \mathrm{S}_{m+1+n}(M,N)^m & =_{\mathrm{df}} & \mathrm{S}_{m+n}(M^m,N^m) \end{array}$$

Lemma 5.15 *If m is free in M then:*

$$\begin{array}{rcl} \mathrm{K}_m(M^m) & \rhd^* & M \\ \mathrm{S}_m(M,N) & \rhd^* & M^m \\ \mathrm{S}_{m+1}(M,\mathrm{I}_m) & = & M^m \end{array}$$

Lemma 5.16 *If $x \notin \mathrm{FV}(M)$ then 0 is free in $[x]M$ and*

$$([x]M)^0 \quad = \quad M$$

Corollary 5.17 (η-equality) *The η-equality rule is sound.*

6 Logical Connectives and Inductive Types

We can also formalize the logical connectives or inductive types as usual in sequent calculus. We give typing rules and equalities for the inductive types 0, 1, $A \times B$ and $A + B$, corresponding to the logical propositions absurdity, truth, conjunction and disjunction.

$$(\mathrm{R}^0) \qquad \frac{}{\vdash_\Gamma \mathrm{R}^0_{|X|} : (X,0) \to A}$$

$$(*) \qquad \frac{}{\vdash_\Gamma *_{|X|} : X \to 1}$$

$$(\mathrm{R}^1) \qquad \frac{\vdash_\Gamma M : X \to A}{\vdash_\Gamma \mathrm{R}^1_{|X|}(M) : (X,1) \to A}$$

$$(I) \qquad \frac{\vdash_\Gamma M : X \to A}{\vdash_\Gamma \mathrm{i}_{|X|}(M) : X \to A+B}$$

$$(J) \qquad \frac{\vdash_\Gamma M : X \to B}{\vdash_\Gamma \mathrm{j}_{|X|}(M) : X \to A+B}$$

$$(\mathrm{R}^+) \qquad \frac{\vdash_\Gamma M : X, A \to C \quad \vdash_\Gamma N : X, B \to C}{\vdash_\Gamma \mathrm{R}^+_{|X|}(M,N) : (X, A+B) \to C}$$

$$(Pair) \qquad \frac{\vdash_\Gamma M : X \to A \quad \vdash_\Gamma N : X \to B}{\vdash_\Gamma \mathrm{p}_{|X|}(M,N) : X \to A \times B}$$

$$(\mathrm{R}^\times) \qquad \frac{\vdash_\Gamma M : (X,A,B) \to C}{\vdash_\Gamma \mathrm{R}^\times_{|X|}(M) : (X, A \times B) \to C}$$

There is also an equality corresponding to the removal of the non-canonical (elimination) constant, plus the obvious extensions of the β- and ξ-equalities.

$$\begin{array}{rcl}
\mathrm{S}_m(\mathrm{R}^1_m(M), *_m) & = & M \\
\mathrm{S}_m(\mathrm{R}^+_m(M,N), \mathrm{i}_m(P)) & = & \mathrm{S}_m(M,P) \\
\mathrm{S}_m(\mathrm{R}^+_m(M,N), \mathrm{j}_m(P)) & = & \mathrm{S}_m(N,P) \\
\mathrm{S}_m(\mathrm{R}^\times_m(M), \mathrm{p}_m(N,P)) & = & \mathrm{S}_m(\mathrm{S}_m(M,N),P)
\end{array}$$

7 Explicit Substitution

In this section we outline a possible extension of sequent combinators to deal explicitly with the isomorphism between $A \times B \Rightarrow C$ and $A \Rightarrow B \Rightarrow C$. We leave the technical details of such a system as future work.

We notice that the typing rules for the sequent combinators I_m, $\mathrm{K}_m(M)$ and $\mathrm{S}_m(M,N)$ are analogous to the rules (*Var*), (*Thin*) and (*Subst*) of the simply-typed lambda calculus. This leads us to consider an extension of sequent combinators including constructors corresponding to the rules of inference of the simply-typed lambda calculus. We shall have the judgement form $\vdash_\Gamma M : X \to A$, where in this system the $\to$ is part of the judgement form rather than a metatheoretic operation as in Section 3. We add the new term constructors M^*_m and $\mathrm{app}_m(M,N)$ to the terms of sequent combinators.

$$(Par) \qquad \frac{x{:}A \in \Gamma}{\vdash_\Gamma x : \ () \to A}$$

$$(\mathrm{I}) \qquad \frac{}{\vdash_\Gamma \mathrm{I}_{|X|} : \ (X, A) \to A}$$

$$(\mathrm{K}) \qquad \frac{\vdash_\Gamma M : \ X \to A}{\vdash_\Gamma \mathrm{K}_{|X|}(M) : \ (X, B) \to A}$$

$$(\mathrm{S}) \qquad \frac{\vdash_\Gamma M : \ (X, A) \to B \quad \vdash_\Gamma N : \ X \to A}{\vdash_\Gamma \mathrm{S}_{|X|}(M, N) : \ X \to B}$$

$$(*) \qquad \frac{\vdash_\Gamma M : \ (X, A) \to B}{\vdash_\Gamma M^*_{|X|} : \ X \to A \Rightarrow B}$$

$$(\mathrm{app}) \qquad \frac{\vdash_\Gamma M : \ X \to A \Rightarrow B \quad \vdash_\Gamma N : \ X \to A}{\vdash_\Gamma \mathrm{app}_{|X|}(M, N) : \ X \to B}$$

The rules of inference are in the figure, where we again understand that Γ is a context for each rule.

To the equalities for sequent combinators we add the following new equalities:

$$\begin{array}{rcl} \mathrm{app}_m(M^*_m, N) & = & \mathrm{S}_m(M, N) \\ \mathrm{S}_m(M^*_{m+1+n}, N) & = & (\mathrm{S}_m(M, N))^*_{m+n} \\ \mathrm{S}_m(\mathrm{app}_{m+1+n}(M, N), P) & = & \mathrm{app}_{m+n}(\mathrm{S}_m(M, P), \mathrm{S}_m(N, P)) \end{array}$$

The informal meanings of the indexed constants I_m, $\mathrm{K}_m(M)$ and $\mathrm{S}_m(M, N)$ in this system are different to the informal meanings of the constants in sequent combinators. Here, we understand I_m as the de Bruijn variable 0, $\mathrm{K}_m(M)$ as an analogue of the lifting operation, and $\mathrm{S}_m(M, N)$ as substitution.

The abstraction algorithm, Definition 5.1, and textual substitution, Definition 5.3, can be adapted to this system straightforwardly. Functional Completeness, Proposition 5.6, now expresses the coherent relationship between object and metatheoretic substitution. We can define named lambda abstraction and application in the empty context () as follows:

$$\lambda x.M \quad =_{\mathrm{df}} \quad ([x]M)^*_0$$

As we have mentioned, this system is very close to a de Bruijn system with explicit substitution and lifting. Such a system has already been presented by Kamareddine and Ríos [15]. Our presentation is slightly simpler because we consider de Bruijn variables n represented explicitly as sequences of thinnings over the identity, i.e. $\mathrm{K}_m(\ldots \mathrm{K}_m(\mathrm{I}_m))$, leading to simpler reduction rules. More importantly, while their work considers meta-variables, they do not consider the abstraction algorithm or the relationship between object and metatheoretic substitution. This means that they cannot define named lambda abstraction or

show results about α-equivalent terms. Similarly, they do not recognize the similarity of their system with combinatory logic or Hilbert systems, nor do they compare their work with categorical logic. We feel that putting the work in its proper historical context, and addressing established questions about object language and meta-language, are essential to the widespread acceptance of such a system as a logical foundation in addition to a basis for implementations.

8 Further Work

The most significant outstanding conjecture for the systems is strong normalization, which would imply confluence. If this property holds then we believe that the clearer relationship between the metatheoretic and object substitutions will simplify the models necessary for such a proof. In particular, Kripke models might only need to vary with respect to the metatheoretic context, avoiding the complex Kripke interpretation of functional types. This would lead to a proof that is simpler in some aspects than proofs of strong normalization for traditional lambda calculi [10, 11], especially when considered for systems of dependent types [4].

Although the form of our judgements and rules is intuitively similar to the sequent calculus, the actual rules are different, instead resembling the rules for the combinators S and K. A calculus following the sequent calculus more closely will be the study of further work. This also leads naturally to the study of linear systems.

Explicit substitution leads to a cleaner treatment of unification [8], and Dougherty [7] has used combinators as the basis for higher-order unification. We believe that our calculus could be used in a similar way. Combined with an extension to dependent types, this would lead to a more formal foundation for proof development in type-theory based proof assistants.

Extending our approach to dependent types could also lead to simpler implementations of proof checkers. We also hope to be able to incorporate functional programming techniques in implementing normalization procedures for type theory, which has been impossible so far because any techniques in functional programming based on combinators do not capture the rule ξ.

9 Conclusions

In the introduction, we mentioned several disadvantages of Hilbert systems when compared with natural deduction systems. We can now see how these disadvantages are addressed by sequent combinators:

- The size of sequent combinators does not grow substantially under the abstraction algorithm. The causes for growth are $[x]y$, which introduces a new term constructor, and the index for each of the combinator term constructors. As Curien notes [5], if

we suppose that the indices are encoded in binary then abstraction increases the size by $n \log n$.

- We have showed that our system behaves well with respect to the metatheoretic operations of abstraction and substitution.
- We have proved (Corollary 5.9 and Theorem 5.10) that the reduction and equality for lambda calculus is preserved by translation into sequent combinators.
- We have demonstrated in Section 6 that the duality between introduction and elimination of natural deduction also exists in sequent combinators.

We therefore believe that we have demonstrated that Hilbert systems can be a viable alternative to lambda calculi.

Section 7 gives a further refinement of the notion of Hilbert system. It shows that a particular system of explicit substitution can be considered as a Hilbert system, where the object notion of substitution is coherent with the metatheoretic one.

Acknowledgments

Thanks to Rod Burstall, Adriana Compagnoni, Peter Hancock, Martin Hyland, Zhaohui Luo, James McKinna and Gordon Plotkin for encouragement and useful comments. The work reported in this paper was in part carried out at the University of Cambridge Computer Laboratory.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] A. Church. *Introduction to Mathematical Logic*. Princeton University Press, 1956.

[4] T. Coquand and J. Gallier. A proof of strong normalization for the theory of constructions using a Kripke-like interpretation. In *Workshop on Logical Frameworks–Preliminary Proceedings*, 1990.

[5] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman, second edition, 1993. Available from John Wiley and Sons.

[6] H. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[7] D. J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2):273–298, 1993.

[8] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions (extended abstract). In *LICS Proceedings*, pages 366–374, 1995.

[9] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Works of Gerhard Gentzen*. North Holland, Amsterdam, 1969.

[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[11] H. Goguen. Typed operational semantics. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer–Verlag, 1995.

[12] H. Goguen and Z. Luo. Inductive data types: Well-ordering types revisited. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 198–218. Cambridge University Press, 1993.

[13] J. Hughes. Supercombinators, a new implementation method for applicative languages. *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

[14] T. Johnsson. Efficient compilation of lazy evaluation. *Proceedings of the ACM Conference on Compiler Construction*, pages 58–69, 1984.

[15] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 1996. to appear.

[16] R. Kennaway and R. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, Oct. 1988.

[17] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge studies in advanced mathematics. Cambridge University Press, 1989.

[18] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. C. Shepherdson, editors, *Logic Colloqium 1973*, pages 73–118, 1975.

[19] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, London, 1986.

[20] D. A. Turner. A new implementation technique for applicative languages. *Software–Practice and Experience*, (9):31–49, 1979.