Computational interpretation of classical logic

A Formulae-as-Types Notion of Control

Timothy G. Griffin*
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The programming language Scheme contains the control construct `call/cc` that allows access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps. We show that the well-known formulae-as-types correspondence, which relates a constructive proof of a formula α to a program of type α, can be extended to a typed Idealized Scheme. What is surprising about this correspondence is that it relates *classical* proofs to typed programs. The existence of computationally interesting "classical programs" — programs of type α, where α holds classically, but not constructively — is illustrated by the definition of conjunctive, disjunctive, and existential types using standard classical definitions. We also prove that all evaluations of typed terms in Idealized Scheme are finite.

1 Introduction

The formulae-as-types correspondence [10,18,8], also referred to as the propositions-as-types correspondence and as the Curry/Howard isomorphism, relates a constructive proof of a formula α to a program of type α. This correspondence has been restricted to constructive logic because it is widely believed that, in general, classical proofs lack computational content. This paper shows, however, that the formulae-as-types correspondence *can* be extended to classical logic in a computationally interesting way. It is shown that classical proofs posses computational content when the notion of computation is extended to include explicit access to the current control context.

This notion of computation is found in the programming language Scheme [16], which contains the control construct `call/cc`[1] that provides access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps, and allows for programs that are more efficient than purely functional programs. The formulae-as-types correspondence presented in this paper is based on a typed version of *Idealized Scheme* — a typed ISWIM containing an operator $\mathcal{C}$ similar to `call/cc` — developed by Felleisen *et al* [3,2,4] for reasoning about Scheme programs.

Section 2 reviews ISWIM and its extension to Idealized Scheme (IS) with the control operator $\mathcal{C}$ of Felleisen *et al.* Roughly speaking, the evaluation of $\mathcal{C}(M)$ abandons the current control context and applies M to a procedural abstraction of this context.

A typed version of Idealized Scheme is presented in Section 3 together with a formulae-as-types correspondence between typed terms and natural deduction proofs for classical implicational logic. Types include the type $\perp$, which corresponds to the proposition "false." The type $\alpha \to \perp$ is abbreviated as $\neg\alpha$ ("not α"). An application of $\mathcal{C}$ is typed as follows. If M is of type $\neg\neg\alpha$, then $\mathcal{C}(M)$ is of type α. This rule corresponds to the classical inferrence rule for elimination of double negation.

Section 4 demonstrates that there are computationally interesting typed IS programs of type α, where α holds classically, but not constructively. It is shown that if conjunctive, disjunctive, and existential types are defined using standard classical definitions, then

*This work was supported in part by DARPA grant CCR-87-20277. The author's current address: *Departamento de Ciência da Computação, IMECC - UNICAMP, Caixa Postal 6065, 13801 Campinas SP, Brazil.* email: griffin@bruc.ansp.br

[1] `call/cc` abbreviates `call-with-current-continuation`.

the operations of pairing, projection, injection, and analysis by cases can be defined using $\mathcal{C}$.

There are many equivalent ways of defining classical logic. For example, in place of double negation elimination, classical logic is often defined by adding the law of the excluded middle, $\alpha \vee \neg\alpha$, to constructive logic. Section 5 shows that the law of the excluded middle can be given an operational interpretation that is computationally equivalent to that of $\mathcal{C}$.

In Section 6 it is shown that the well-known cps (continuation passing style) transform corresponds to an embedding of classical into constructive logic. Section 7 uses a modified cps transform to prove that all evaluations of well-typed IS programs are finite.

2 From ISWIM to Idealized Scheme

Landin's ISWIM [11,12] is a call-by-value language whose core syntax is made up of expressions of the λ-calculus,

$$N \quad ::= \quad x \quad | \quad NN \quad | \quad \lambda x.N$$

where x ranges over an infinite set of variables. The operational semantics of ISWIM was defined by Landin in terms of the SECD-machine. Plotkin [14] showed that this definition is equivalent[2] to the (partial) function eval_v:

1. $\mathrm{eval}_v(V) = V$,
2. $\mathrm{eval}_v(MN) = \mathrm{eval}_v(Q[V/x])$ if $\mathrm{eval}_v(M) = \lambda x.Q$ and $\mathrm{eval}_v(N) = V$.

Each V represents a *value*, where values are defined to be variables or λ-abstractions. Throughout this paper the metavariables V, V_1, V_2, ... will range over values. The notation $M[N/x]$ denotes the usual capture-avoiding substitution of N for all free occurrences of x in M.

An expression of the form $(\lambda x.M)V$ is called a β_v-redex. The function eval_v reduces at each step the leftmost-outermost β_v-redex not inside the scope of a λ-abstraction. Felleisen *et al* [3,4] have formalized this evaluation order in terms of *evaluation contexts*. ISWIM evaluation contexts E are defined inductively as

$$E \quad ::= \quad [\,] \quad | \quad EN \quad | \quad VE,$$

where [ ] represents a "hole." If E is an evaluation context, then $E[M]$ denotes the term that results from placing M in the hole of E. It is not difficult to show that any closed term M is either a value or can be written in a *unique* way as $M = E[R]$, where R is a β_v-redex. Moreover, R is the leftmost-outermost β_v-redex of M that is not inside of a λ-abstraction. The notation $M \propto E[R]$ means that $E[R]$ is this unique representation of M. For example, if $E_0 = (\lambda x, M)[\,]$ and $E_1 = [\,]$, then

$$(\lambda x, M)V = E_0[V] \propto E_1[(\lambda x.M)V].$$

The unique representation of any non-value in terms of an evaluation context and a β_v-redex gives rise to the context rewrite rule

$$E[(\lambda x.M)V] \longmapsto_{\beta_v} E[M[V/x]], \qquad (\longmapsto_{\beta_v})$$

whose reflexive, transitive closure $\longmapsto^*_{\beta_v}$ is equivalent to eval_v.

Lemma 1 $\mathrm{eval}_v(M) = V$ *iff* $M \longmapsto^*_{\beta_v} V$.

In other words, $\longmapsto_{\beta_v}$ can be taken as defining an abstract operational semantics for ISWIM. An ISWIM term M *evaluates* to V if and only if $M \longmapsto^*_{\beta_v} V$.

The notation of evaluation contexts gives a clear picture of the manner in which subterms are evaluated during the evaluation of a term. (The notation $\longmapsto^k_{\beta_v}$ denotes a k-fold application of the $\longmapsto_{\beta_v}$ rule.)

Lemma 2 *1. If* $E[M] \longmapsto^k_{\beta_v} E[N]$, *then* $M \longmapsto^k_{\beta_v} N$.

2. If $E[M] \longmapsto^*_{\beta_v} V$, *then there is a value* V_0 *such that* $E[M] \longmapsto^* E[V_0] \longmapsto^*_{\beta_v} V$.

Thus, at any point i in an evaluation sequence

$$M_0 \longmapsto_{\beta_v} M_1 \longmapsto_{\beta_v} \cdots \longmapsto_{\beta_v} M_i \longmapsto_{\beta_v} \cdots$$

if $M_i = E[N]$, for a non-value N, then E must "wait" for N to evaluate to a value before the evaluation sequence can continue with computations involving subterms of E. That is, E represents *the rest of the computation that remains to be done after N is evaluated.* The context E is called the *continuation* (or *control context*) of N at this point in the evaluation sequence. The notation of evaluation contexts allows, as we shall see below, a concise specification of the operational semantics of operators that manipulate continuations (indeed, this was its intended use [3,2,4,1]).

The programming language Scheme [16] contains **call/cc**, a control construct that provides programs with direct access to a procedural abstraction representing the current continuation (the current control context). Felleisen *et al* [3,2,4,1] have presented an extension to ISWIM called Idealized Scheme[3], or IS,

[2]This paper ignores constants and their evaluation.

[3]This paper treats only the assignment-free sublanguage of Idelaized Scheme.

which incorporates two constructs that manipulate control contexts. IS expressions are defined by extending the grammar of ISWIM as follows:

$$N \ ::= \ \cdots \ \mathcal{A}(N) \ \mid \ \mathcal{C}(N).$$

The operators $\mathcal{A}$ and $\mathcal{C}$ are called, respectively, *abort* and and *control*. In IS, any closed term M is either a value, or can be written in a unique way as $M = E[R]$, where R is either a β_v-redex, $R = \mathcal{A}(N)$, or $R = \mathcal{C}(N)$.

Informally, the evaluation of $\mathcal{A}(M)$ throws away the current control context and continues with the evaluation of M. This is expressed with a context rewrite rule, where the definition of evaluation contexts has been extended to IS expressions in the obvious way, as

$$E[\mathcal{A}(M)] \ \longmapsto_{\mathcal{A}} \ M \qquad (\longmapsto_{\mathcal{A}})$$

The operational semantics of $\mathcal{C}(M)$ can be described informally as follows. As with $\mathcal{A}$, the evalation of $E[\mathcal{C}(M)]$ abondons the control context E. The term M is then applied to a procedural abstraction of the abandoned control context. If this procedure is invoked with a value V in any context E_1, then E_1 is abandoned and evaluation resumes with $E[V]$. This is expressed with the rule,

$$E[\mathcal{C}(M)] \ \longmapsto_{\mathcal{C}} \ M\lambda z.\mathcal{A}(E[z]). \qquad (\longmapsto_{\mathcal{C}})$$

The operator $\mathcal{A}$ can be defined in terms of $\mathcal{C}$ as

$$\mathcal{A}(M) \ \stackrel{\text{def}}{=} \ \mathcal{C}(\lambda d.M),$$

where d is a dummy variable not free in M, since

$$\begin{array}{rcl} E[\mathcal{A}(M)] & = & E[\mathcal{C}(\lambda d.M)] \\ & \longmapsto_{\mathcal{C}} & (\lambda d.M)\lambda z.\mathcal{A}(E[z]) \\ & \longmapsto_{\beta_v} & M \end{array}$$

Therefore, $\mathcal{A}(M)$ will be treated as a defined construct, and the rules $\longmapsto_{\beta_v}$ and $\longmapsto_{\mathcal{C}}$ will be treated as defining the operational semantics of IS. The notation $\longmapsto_u$ denotes the union of the two evaluation rules.

The operational semantics of $\mathcal{C}$ differs from that of `call/cc` in that $\mathcal{C}$ need not return to the location of its use. If a version of `call/cc` were to be added to IS, say $\mathcal{K}$, then it would have the evaluation rule

$$E[\mathcal{K}(M)] \ \longmapsto_{\mathcal{K}} \ E[M\lambda z.\mathcal{A}(E[z])]. \qquad (\longmapsto_{\mathcal{K}})$$

However, this addition is not necessary since an operator computationally equivalent to $\mathcal{K}$ can be defined as

$$\mathcal{K}_d(M) \ \stackrel{\text{def}}{=} \ \mathcal{C}(\lambda k.k(Mk)). \qquad (\mathcal{K}_d)$$

One use of $\mathcal{K}_d$ is in the implemention of a "catch/throw" mechanism similar to that of Common Lisp [17]. Think of the evaluation of $E_0[\mathcal{K}_d(\lambda j.M)]$ as a "catch" that labels the current context with the name j. If j is never invoked, or "thrown to" during the evaluation of M, then this expression returns "normally." If, on the other hand, an application of j, such as $E_1[jV]$, is encountered during the evaluation of M, then the value V is "thrown back to" the location labeled by j. That is, the context E_1 is abandoned and evaluation resumes with $E_0[V]$. The following illustrates how this is accomplished with the evaluation rules of Idealized Scheme. If $Q = \lambda z.\mathcal{A}(E_0[z])$, then

$$\begin{array}{rcl} E_0[\mathcal{K}_d(\lambda j.M)] & \longmapsto_{\mathcal{C}} & (\lambda k.k((\lambda j.M)k))Q \\ & \longmapsto_{\beta_v} & Q((\lambda j.M)Q) \\ & \longmapsto_{\beta_v} & Q(M[Q/j]) \end{array}$$

If $M[Q/j] \longmapsto^*_{\beta_v} V$, then the evaluation returns "normally" with

$$\begin{array}{rl} \longmapsto^*_{\beta_v} & QV \\ \longmapsto_{\beta} & \mathcal{A}(E_0[V]) \\ \longmapsto_{\mathcal{A}} & E_0[V] \\ \cdots & \cdots \end{array}$$

If, on the other hand, a value is eventually thrown, then

$$\begin{array}{rcl} Q(M[Q/j]) & \longmapsto^*_{\beta_v} & E_1[QV] \\ & \longmapsto_{\beta_v} & E_1[\mathcal{A}(E_0[V])] \\ & \longmapsto_{\mathcal{A}} & E_0[V] \\ & \cdots & \cdots \end{array}$$

showing that the context E_1 is abandoned and that evaluation continues with V in the restored context E_0.

3 Formulae-as-types for IS

This section develops a typed version of Idealized Scheme (IS$_t$) together with a formulae-as-types correspondence between IS$_t$ expressions and a system of natural deduction for classical implicational logic. The evaluation of typed terms requires a minor modification to the operational semantics of IS.

Define type expressions α as

$$\alpha \ ::= \ t \ \mid \ \alpha \rightarrow \alpha',$$

where t is a member of a set of atomic types. Type expressions will also be read as propositions (formulae), with $\alpha \rightarrow \beta$ representing "α implies β."

The syntax of ISWIM is modified so that variables are tagged with a type expression: x^α and $\lambda x^\alpha.M$.

Typed ISWIM, written as ISWIM$_t$, is defined in the same way as the simply-typed λ-calculus. A variable x^α has type α; if M has type $\alpha \to \beta$ and N has type α, then MN has type β; if M has type β, then $\lambda x^\alpha.M$ has type $\alpha \to \beta$. The notation M^α means that M has type α.

First, the Curry-Howard isomorphism between ISWIM$_t$ terms and natural deduction proofs for minimal logic (M) is presented. The reader is referred to Prawitz [15], Stenlund [18], and Girard [8], for a complete treatment. Second, the correspondence is extended to IS$_t$ with a logically consistent typing for $\mathcal{C}$.

Natural deduction derivations (proofs) Σ are tree-structured objects whose leaves contain formulae representing assumptions and whose nodes represent the application of inference rules. A derivation Σ with conclusion α is written as

$$\begin{array}{c}\Sigma\\ \alpha\end{array}$$

The system **M** of natuarl deduction derivations is generated from assumptions α, the inference rule for $\to$-elimination ($\to E$, or *modus ponens*)

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ \alpha \to \beta & \alpha\end{array}}{\beta}$$

and the inference rule for $\to$-introduction ($\to I$)

$$\frac{\begin{array}{c}[\alpha]\\ \Sigma\\ \beta\end{array}}{\alpha \to \beta}$$

The notation

$$\begin{array}{c}\alpha\\ \Sigma\\ \beta\end{array}$$

means that there are zero or more undischarged occurrences of the assumption α in the derivation Σ, while the notation

$$\begin{array}{c}[\alpha]\\ \Sigma\\ \beta\end{array}$$

means that some of these assumptions have been discharged (made unavailable).

For each derivation Σ there is a corresonding ISWIM$_t$ term M of type α, which is defined by induction on the structure of Σ. Assume that the assumptions of Σ are divided into a disjoint collection of sets, each associated with a unique variable. An assumption α corresponds to the variable x^α, where x is the variable associated with the set for α. If

$$\begin{array}{c}\Sigma_1\\ \alpha \to \beta\end{array}$$

corresponds to the term $M^{\alpha\to\beta}$ and

$$\begin{array}{c}\Sigma_2\\ \alpha\end{array}$$

corresponds to the term N^α, then

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ \alpha \to \beta & \alpha\end{array}}{\beta}$$

corresponds to $(MN)^\beta$. If

$$\begin{array}{c}\alpha\\ \Sigma\\ \beta\end{array}$$

corresponds to M^β, then

$$\frac{\begin{array}{c}[\alpha]\\ \Sigma\\ \beta\end{array}}{\alpha \to \beta}$$

corresponds to $(\lambda x^\alpha.M)^{\alpha\to\beta}$, provided that the set of discharged assumptions is the set associated with the variable x.

We will now extend the correspondence between typed terms and proofs to IS by finding a logically consistent typing for $\mathcal{C}$. Let us start by looking at the $\mapsto_\mathcal{C}$ rule

$$E[\mathcal{C}(M)] \mapsto_\mathcal{C} M\lambda z.\mathcal{A}(E[z]). \qquad (\mapsto_\mathcal{C})$$

Let α and β be arbitrary types. Suppose that E is of type β and that the hole in E is expecting a term of type α. It seems reasonable to give the term $\lambda z.\mathcal{A}(E[z])$ the type $\alpha \to \beta$ since for any value V of type α,

$$(\lambda z.\mathcal{A}(E[z]))V \mapsto_u^+ E[V],$$

which is of type β. Therefore, since both sides of the $\mapsto_\mathcal{C}$ rule are of type β, M must have type $(\alpha \to \beta) \to \beta$. We then arrive at the following typing rule for $\mathcal{C}(M)$: if M has type $(\alpha \to \beta) \to \beta$, then $\mathcal{C}(M)$ has type α.

It follows from this derivation that if N is a closed term of type β, then $\mathcal{A}(N) = \mathcal{C}(\lambda d.N)$ can be given *any* type α. Therefore, if we want a type system that is logically consistent when types are read as propositions, β must be a proposition that has no proof (otherwise every proposition is provable). Assume that the set of atomic types contains the type $\bot$, which represents an empty type, or the proposition "false." Define $\neg\alpha$ (read "not α") as

$$\neg\alpha \overset{\text{def}}{=} \alpha \to \bot, \qquad (\neg\alpha).$$

We then arrive at a logically consistent typing for $\mathcal{C}(M)$: if M has type $\neg\neg\alpha$, then $\mathcal{C}(M)$ has type α. This will be the typing used for typed Idealized Scheme, which is written as IS_t. Such an instance of $\mathcal{C}(M)$ will often be written as $\mathcal{C}^\alpha(M)$ in order to make explicit the type of the term.

From a logical perspective, $\mathcal{C}^\alpha(M)$ correponds to the *classical* proof rule for double negation elimination ($\bot_c$)

$$\frac{\begin{array}{c}\Sigma\\ \neg\neg\alpha\end{array}}{\alpha}$$

if $M^{\neg\neg\alpha}$ corresponds to the derivation Σ. The system **C** is defined to be **M** extended with the $\bot_c$ rule.

Note that $\mathcal{A}(M)$ now corresponds to the constructive rule for $\bot$-elimination ($\bot_e$)

$$\frac{\begin{array}{c}\Sigma\\ \bot\end{array}}{\alpha}$$

which can be derived in **C**. The notation $\mathcal{A}^\alpha(M)$ indicates that this term has type α. The constructive system **J** is defined to be **M** extended with the $\bot_e$ rule.

There is one problem with this typing of IS. The $\longmapsto_{\mathcal{C}}$ rule applies only when the entire expression $E[\mathcal{C}(M)]$ is of type $\bot$, and since there are no closed terms of this type, the rule is useless! To rectify this problem, a minor modification is made to the operational semantics of IS. The basic idea is as follows. Instead of evaluating an expression M^α with the $\longmapsto_u$ rules, the expression $\mathcal{C}(\lambda k^{\neg\alpha}.kM)$ is evaluated with the rules of $\longmapsto_u$ being applied only inside of the expression $\mathcal{C}(\lambda k.\cdots)$. The rules now make "type sense" since the body of the λ-expression is of type $\bot$.

Formally, define the operational semantics $\longmapsto_t$ as the union of the following rules.

$$\begin{array}{rcl} \mathcal{C}(\lambda k.E[(\lambda x.M)V]) & \longmapsto_{t\beta_v} & \mathcal{C}(\lambda k.E[M[V/x]]) \\ \mathcal{C}(\lambda k.E[\mathcal{C}(M)]) & \longmapsto_{t\mathcal{C}} & \mathcal{C}(\lambda k.M\lambda z.\mathcal{A}(E[z])) \\ \mathcal{C}(\lambda k.kV) & \longmapsto_{\mathcal{C}_e} & V \end{array}$$

The last rule is subject to the proviso that k is not free in V. This rule merely allows for the removal of the outermost $\mathcal{C}$ at the end of some computations. An expression is in $\longmapsto_t$ normal form if none of these rules apply.

Definition 1 (evaluation of typed terms) *A closed* IS_t *expression* M^α *evaluates to* Q *if*

$$\mathcal{C}^\alpha(\lambda k^{\neg\alpha}.kM) \longmapsto_t^* Q$$

and Q *is in* $\longmapsto_t$ *normal form.*

That $\longmapsto_t$ is only a minor modification to the $\longmapsto_u$ semantics is stated in the following lemma.

Lemma 3 *If* $\mathcal{C}(\lambda k.kM) \longmapsto_u^* V$*, then* $\mathcal{C}(\lambda k.kM) \longmapsto_t^* Q$*, where* Q *is either* V*,* $\mathcal{C}\lambda k.kV'$*, or* $\mathcal{C}\lambda k.V'$*, and* $V = V'[\lambda x.\mathcal{A}(x)/k]$.

In other words, the only type violation of the system $\longmapsto_u$ is the replacement of the top-level continuation k with $\lambda x.\mathcal{A}(x)$.

The types of "classical programs" cannot be given the same operational interpretation as the types of "constructive programs." A program M corresponding to a constructive proof of $\alpha \to \beta$ takes inputs of type α to outputs of type β. This is no longer the case with classical programs since the evaluation of an expression need not return to the point of its evaluation but may "jump" to some other evaluation context. In the type system presented here, the distinction between a "returning expression" and a "jumping expression" cannot be made by inspecting an expression's type. Thus, if M is a classical program of type $\alpha \to \beta$ and N is a classical program of type α, we know only that if the application of M to N returns to the current control context, then it will return with a (classical) value of type β. Note that the evaluation of either M, N, or the application of M to N could result in a jump.

4 Conjunctive, disjunctive, and existential types

This section demonstrates that there are computationally interesting IS_t terms of type α, where α holds in classical, but not constructive, logic. It is shown that if conjunctive and disjunctive types are defined using standard classical definitions, then the operations of pairing, projection, injection, and analysis by cases can be defined using $\mathcal{C}$. The section concludes by pointing out that if IS_t types are extended with universal types $\forall x^t.\alpha(x)$, then existential types $\exists x^t.\alpha(x)$ can be defined in IS_t.

That the connectives for conjunction and disjunction cannot be defined in constructive (implicational) logic is related, via the Curry/Howard correpondence, to the fact that pairing, projection, injection, and analysis by cases are not definable in the simply typed λ-calculus. It is well known, however, that the connectives for conjunction and disjunction can be defined *classically* in terms of negation and implication as

$$\begin{aligned} \alpha \wedge \beta &\overset{\text{def}}{=} \neg(\alpha \to \neg\beta), \\ \alpha \vee \beta &\overset{\text{def}}{=} \neg\alpha \to \beta. \end{aligned}$$

The remainder of the section proceeds as follows. The introduction and elimination rules for $\wedge$ and $\vee$ are derived in the classical system **C** and the computational properties of the IS_t terms corresponding to these derived rules are investigated. It is shown that these terms can be used for pairing, projection, injection, and analysis by cases.

The $\wedge$-introduction rule

$$\dfrac{\overset{\Sigma_1}{\alpha} \quad \overset{\Sigma_2}{\beta}}{\alpha \wedge \beta} \quad (\wedge I)$$

can be derived in **C** as

$$\dfrac{\dfrac{\dfrac{[\alpha \to \neg\beta] \quad \overset{\Sigma_1}{\alpha}}{\neg\beta} \qquad \overset{\Sigma_2}{\beta}}{\bot}}{\neg(\alpha \to \neg\beta)}$$

If M^α and N^β are IS_t terms corresponding to the derivations Σ_1 and Σ_2, then the IS_t term

$$\langle M, N\rangle \overset{\text{def}}{=} \lambda f^{\alpha \to \neg\beta}.fMN$$

of type $\alpha \wedge \beta$ corresponds to the derived $\wedge$-introduction rule.

The two rules for $\wedge$-elimiantion

$$\dfrac{\overset{\Sigma}{\alpha_1 \wedge \alpha_2}}{\alpha_i} \quad (\wedge E_i),$$

can be derived in **C** as

$$\dfrac{\dfrac{\dfrac{\overset{\Sigma}{\neg(\alpha_1 \to \neg\alpha_2)} \qquad \dfrac{\dfrac{\dfrac{[\alpha_i] \quad [\neg\alpha_i]}{\bot}}{\neg\alpha_2}}{\alpha_1 \to \neg\alpha_2}}{\bot}}{\neg\neg\alpha_i}}{\alpha_i}$$

If the term M of type $\alpha \wedge \beta$ corresponds to the derivation to Σ, then the IS_t term

$$\pi_i(M) \overset{\text{def}}{=} \mathcal{C}(\lambda j^{\neg\alpha_i}.M\lambda x_1^{\alpha_1}.\lambda x_2^{\alpha_2}.jx_i)$$

of type α_i corresponds to the derived rule for $\wedge$-elimination.

Computationally, the terms $\langle M, N\rangle$ and π_i represent operations of pairing and projection. The derivations of the computational properties of these terms are carried out with the $\mapsto_u$ rules, with the understanding that typed terms are to be evaluated using the $\mapsto_t$ rules. This is done only to avoid the notational clutter of wrapping around each term the expression $\mathcal{C}(\lambda k.\cdots)$.

Let $Q = \lambda z.\mathcal{A}(E[z])$, then

$$\begin{aligned} E[\pi_1(\langle M_1, M_2\rangle)] &\mapsto_{\mathcal{C}} \langle M_1, M_2\rangle(\lambda x_1.\lambda x_2.Qx_i) \\ &\mapsto_{\beta_v} (\lambda x_1.\lambda x_2.Qx_i)M_1M_2 \end{aligned}$$

Now, if both M_1 and M_2 evaluate to values V_1 and V_2, respectively, then the evaluation has the form

$$\begin{aligned} &\mapsto_u^+ (\lambda x_1.\lambda x_2.Qx_i)V_1M_2 \\ &\mapsto_{\beta_v} (\lambda x_2.Qx_i[V_1/x_1])M_2 \\ &\mapsto_u^+ (\lambda x_2.Qx_i[V_1/x_1])V_2 \\ &\mapsto_\beta QV_i \\ &\mapsto_\beta \mathcal{A}(E[V_i]) \\ &\mapsto_{\mathcal{A}} E[V_i] \\ &\ \vdots \quad \vdots \end{aligned}$$

Thus, the evaluation of $E[\pi_i(\langle M_1, M_2\rangle)]$ forces both M_1 and M_2 to be evaluated to values V_1 and V_2 at the top-level before V_i is thrown back to the context E. Note, however, that in general the terms M_i need not return. As a special case, if the evaluation starts with a pair of values, then we have

$$E[\pi_i(\langle V_1, V_2\rangle)] \mapsto_u^+ E[V_i].$$

This should be campared to adding operators for pairing and projection to ISWIM_t together with the evaluation rule

$$E[\pi_i(\langle M_1, M_2\rangle)] \mapsto_{\pi_i} E[M_i]. \qquad (\mapsto_{\pi_i})$$

If $E[\pi_i(\langle M_1, M_2\rangle)] \mapsto_{\pi_i} E[M_i]$, then M_i must evaluate to a value V_i before the evaluation can continue with subterms of E (by an extension of Lemma 2, Section 2, with the appropriate definition of evaluation contexts). The classical definition requires, however, that *both* M_1 and M_2 are evaluated to values.

Turning to disjunction, the introduction rule

$$\dfrac{\overset{\Sigma}{\alpha_1}}{\alpha_1 \vee \alpha_2} \quad (\vee I_1),$$

can be derived in **C** in such a way that if M^α corresponds to the derivation Σ, then

$$\text{inj}_1(M) \overset{\text{def}}{=} \lambda k^{\neg\alpha_1}.\mathcal{A}^{\alpha_2}(kM)$$

is a IS$_t$ term of type $\alpha_1 \vee \alpha_2$ corresponding to the derived rule for $\vee I_1$. The introduction rule

$$\dfrac{\begin{array}{c}\Sigma\\ \alpha_2\end{array}}{\alpha_1 \vee \alpha_2} \quad (\vee I_2),$$

can be derived in **C** in such a way that if the term M of type α_2 corresponds to the derivation Σ, then

$$\mathrm{inj}_2(M) \overset{\mathrm{def}}{=} \lambda k^{\neg\alpha_1}.M$$

is of type $\alpha_1 \vee \alpha_2$ corresponding to the derived $\vee I_2$-rule. Finally, the $\vee$-elimination rule

$$\dfrac{\begin{array}{c}\Sigma\\ \alpha_1 \vee \alpha_2\end{array} \qquad \begin{array}{c}[\alpha_1]\\ \Sigma_1\\ \delta\end{array} \qquad \begin{array}{c}[\alpha_2]\\ \Sigma_2\\ \delta\end{array}}{\delta} \quad (\vee E).$$

can be derived in **C** in such a way that the term

$$\mathrm{case}(M, F_1, F_2) \overset{\mathrm{def}}{=} \mathcal{C}(\lambda j^{\neg\delta}.j(F_1(M\lambda a.j(F_1 a))))$$

of type δ corresponds to the derived rule when $F_i = \lambda x_i^{\alpha_i}.M_i$ correspond to the derivations

$$\dfrac{\begin{array}{c}[\alpha_i]\\ \Sigma_i\\ \delta\end{array}}{\alpha_i \rightarrow \delta}$$

for $i \in \{1,2\}$.

Computationally, the terms $\mathrm{inj}_i(M)$ and $\mathrm{case}(M, F_1, F_2)$ represent operations of injection and case analysis. Let $Q = \lambda z.\mathcal{A}(E[z])$, then

$$E[\mathrm{case}(\mathrm{inj}_1(N_1), F_1, F_2)] \quad \mapsto_u^+ \quad (\lambda a.Q(F_1 a))N_1,$$

and

$$E[\mathrm{case}(\mathrm{inj}_2(N_2), F_1, F_2)] \quad \mapsto_u^+ \quad Q(F_2 N_2)$$

are easy to derive using the $\mapsto_u$ rules. Suppose that N_i evaluates to V_i. If $F_i V_i$ evaluate to a value V_i', then in both cases evaluation can be continued as

$$\begin{array}{ll} \mapsto_u^+ & QV_i' \\ \mapsto_{\beta_v} & \mathcal{A}(E[V_i']) \\ \mapsto_{\mathcal{A}} & E[V_i'] \\ \vdots & \vdots \end{array}$$

Thus, the result of applying F_i to V_i is evaluated at the top-level to a value V_i' before this value is thrown back to the context E. Again, the evaluation of this application need not return.

Suppose that IS$_t$ types are extended with universal types, $\forall x.\alpha$, where x ranges over integer terms. In logical terms, this corresponds to extending the propositional calculus to a first-order predicate calculus. It is assumed that types (propositions) have been extended to include predicates such as equality. If M has type $\forall x.\alpha$ and n is an integer expression, then Mn has type $\alpha[n/x]$. If x is not free in any type of a free variable of M^α, then $\lambda x.M$ has type $\forall x.alpha$.

Existential types can now be defined with the standard classical definition,

$$\exists x.\alpha \overset{\mathrm{def}}{=} \neg\forall x.\neg\alpha(x).$$

Define the terms

$$P_1 \overset{\mathrm{def}}{=} \lambda x.\lambda w^{\alpha(x)}.\lambda f^{\forall y.\neg\alpha(y)}.fxw$$

of type $\forall x.(\alpha(x) \rightarrow \exists y.\alpha(y))$, and

$$P_2 \overset{\mathrm{def}}{=} \lambda p^{\exists x.\alpha(x)}.\lambda f^{\forall x.(\alpha(x)\rightarrow\beta)}.\mathcal{C}(\lambda j^{\neg\beta}.p(\lambda x.\lambda w.j(fxw)))$$

of type $\exists x.\alpha(x) \rightarrow (\forall x.(\alpha(x) \rightarrow \beta)) \rightarrow \beta$. These terms represent operators for computing with (weak) existential types (see, for example, [10]). For an integer value n, V_1 of type $\alpha[n/x]$, and V_2 of type $\forall x.(\alpha(x) \rightarrow \beta)$, the evaluation

$$E[P_2(P_1 n V_1)V_2] \quad \mapsto_u^+ \quad Q(V_2 n V_1)$$

can be derived with $Q = \lambda z.\mathcal{A}(E[z])$. If $V_2 n V_1$ evaluates to a value V, then this value is thrown back to the context E.

5 The excluded middle

There are many equivalent ways of defining classical logic. For example, in place of double negation elimination, classical logic is often defined by adding the law of the excluded middle, $\alpha \vee \neg\alpha$, to constructive logic. This section shows that the law of the excluded middle can be given an operational interpretation that is computationally equivalent to that of $\mathcal{C}$.

For any α, the law of the excluded middle can be derived in **C** as

$$\dfrac{\dfrac{\dfrac{[\neg(\alpha\vee\neg\alpha)] \qquad \dfrac{\dfrac{\dfrac{[\neg(\alpha\vee\neg\alpha)] \quad \dfrac{[\alpha]}{\alpha\vee\neg\alpha}}{\bot}}{\neg\alpha}}{\alpha\vee\neg\alpha}}{\bot}}{\neg\neg(\alpha\vee\neg\alpha)}}{\alpha\vee\neg\alpha}$$

This derivation corresponds to the IS_t term

$$\mathsf{c}^{\alpha} \stackrel{\text{def}}{=} \mathcal{C}(\lambda j^{\neg(\alpha\vee\beta)}.j(\mathrm{inj}_2(\lambda a^{\alpha}.j(\mathrm{inj}_1(a)))))$$

of type $\alpha \vee \neg\alpha$. It is then easy to derive, using the $\mapsto_u$ rules, an evaluation rule for c,

$$E[\mathsf{c}] \mapsto_{\mathsf{c}} E[\mathrm{inj}_2(\lambda a.Q(\mathrm{inj}_1(a)))],$$

where $Q = \lambda z.\mathcal{A}(E[z])$. (As in the previous section, notational clutter is avoided by using the $\mapsto_u$ evaluation rules.)

Alternatively, suppose that typed constants c^{α} are added to an extended ISWIM, which contains injections and analysis by cases. Note that this corresponds to an alternative formalization of classical logic in which the double negation elimination rule can be derived as

$$\dfrac{\alpha\vee\neg\alpha \qquad [\alpha] \qquad \dfrac{\dfrac{\begin{matrix}\Sigma\\ \neg\neg\alpha \quad [\neg\alpha]\end{matrix}}{\bot}}{\alpha}}{\alpha}$$

This derivation corresponds to the derived version of $\mathcal{C}$,

$$\mathcal{C}^{\alpha}_{\mathsf{c}}(M) \stackrel{\text{def}}{=} \mathrm{case}(\mathsf{c}^{\alpha}, \lambda a^{\alpha}.a, \lambda k^{\neg\alpha}.\mathcal{A}^{\alpha}(Mk)),$$

where M corresponds to Σ. Suppose that $\mapsto_{\mathsf{c}}$ is taken as a primitive evaluation rule and evaluation contexts include contexts of the form $\mathrm{case}(E, M_1, M_2)$. Then the evaluation rule for $\mathcal{C}_{\mathsf{c}}$ can be derived as

$$E[\mathcal{C}_{\mathsf{c}}(M)] \mapsto_{\mathcal{C}_{\mathsf{c}}} M\lambda z.Q'(\mathrm{inj}_1(z)), \qquad (\mapsto_{\mathcal{C}_{\mathsf{c}}})$$

where $Q' = \lambda z.\mathcal{A}(E[\mathrm{case}(z, \lambda a.a, \lambda k.\mathcal{A}(Mk))])$. Note that $\mapsto_{\mathcal{C}_{\mathsf{c}}}$ is computationally equivalent to the $\mapsto_{\mathcal{C}}$ rule, since for any context E_1,

$$E_1[(\lambda z.Q'(\mathrm{inj}_1(z)))V] \mapsto^{+} E[V].$$

Similar results can be obtained for other formalizations of classical logic. For example, suppose classical logic is defined as **J** extended with Peirce's law

$$\dfrac{\begin{matrix}\Sigma\\ (\alpha\to\beta)\to\alpha\end{matrix}}{\alpha}$$

This rule can be put into correspondence with a typed version of $\mathcal{K}$ (see Section 2 for the definition of $\mathcal{K}$) as follows. If M is a term of type $(\alpha \to \beta) \to \alpha$, then $\mathcal{K}^{\alpha}_{\beta}(M)$ has type α. Now $\mathcal{C}$ can then be defined as

$$\mathcal{C}^{\alpha}(M) \stackrel{\text{def}}{=} \mathcal{K}^{\alpha}_{\bot}(\lambda j^{\neg\alpha}.\mathcal{A}^{\alpha}(Mj)),$$

which corresponds to the derivation of double negation elimination using $\bot_e$ and Peirce's law. The $\mapsto_{\mathcal{C}}$ rule can then be derived with the rules $\mapsto_{\beta_v}$, $\mapsto_{\mathcal{A}}$, and $\mapsto_{\mathcal{K}}$.

6 The cps transform is a logical embedding

A common approach to providing a semantics for a language that contains labels and jumps is via a translation to a language that explicitly represents continuations as functions. Such a translation is often called a *continuation passing style* transformation, or simply a *cps* transformation. A cps transform $\overline{M}$ for untyped λ-expressions was introduced by Fischer [7] and extended to expressions containing $\mathcal{C}$ by Felleisen *et al* [3]. A slightly modified cps transform is defined here as

$$\begin{aligned}
\overline{x} &= \lambda k.kx,\\
\overline{\lambda x.M} &= \lambda k.k(\lambda x.\overline{M}),\\
\overline{MN} &= \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk)),\\
\overline{\mathcal{C}(M)} &= \lambda k.\overline{M}(\lambda m.m(\lambda z.\lambda d.kz)\lambda x.\mathcal{A}(x)).
\end{aligned}$$

This definition differs from the one in [3] in the last clause, where we use $\lambda x.\mathcal{A}(x)$ rather than $\lambda x.x$.

Although the cps transform is defined for untyped expressions, it defines a transformation on typed expressions as well. Assume there is a distinguished type o, and define the transformation α^* on types as

$$\begin{aligned}
t^* &= t,\\
(\alpha\to\beta)^* &= \alpha^* \to (\beta^* \to o) \to o.
\end{aligned}$$

Theorem 4 *[cps as a typed transform] If M is an IS_t expression of type α, then $\overline{M}$ has type $(\alpha^* \to o) \to o$.*

This fact simply extends a result of Meyer and Wand [13] from simply-typed terms to typed terms containing $\mathcal{C}$.

An *embedding* of classical implicational logic (**C**) into constructive implicational logic (**J**) is defined to be a translation of formulae α' such that if there is a classical proof of α, then there is a constructive proof of α', where α is classically equivalent to α'. It is interesting to note that if we take $\mathcal{A}$ to be a basic construct, then the cps transform corresponds to such an embedding.

For **S** being **J** or **C**, let $\Gamma \vdash_{\mathbf{S}} \alpha$ represent the assertion that there exists an **S**-derivation for α, all of whose undischarged assumptions are in the set of formulae Γ. Let $\Gamma^* = \{\alpha^* \mid \alpha \in \Gamma\}$. Theorem 4 can now be restated in terms of proofs.

Theorem 5 (cps as a proof transform) *If Σ is a proof of $\Gamma \vdash_{\mathbf{C}} \alpha$ corresponding to M, then there exists a proof $\overline{\Sigma}$ of $\Gamma^* \vdash_{\mathbf{J}} (\alpha^* \to o) \to o$ that corresponds to $\overline{M}$.*

If $o = \bot$, then it is easy to check that for all α,

$$\vdash_{\mathbf{C}} \alpha \leftrightarrow \neg\neg\alpha^*,$$

and so the translation corresponds to an embedding[4].

7 Evaluations are finite

In this section it is shown that all computations with well-typed IS$_t$ terms are finite.

Theorem 6 (finite evaluation) *The evaluation of any well-typed IS$_t$ term M^α is finite.*

The method of proof involves a translation of IS$_t$ terms M to simply-typed λ-terms M' so that any infinite evaluation sequence starting from M induces an infinite β-reduction sequence starting from M'. Then, since there are no infinite β-reductions in the simply-typed λ-calculus (see, for example, [9]) there can be no infinite evaluations of IS$_t$ terms.

An obvious candidate for this translation is the cps transform of the previous section. However, as mentioned in Plotkin [14], the cps transform $\overline{M}$ introduces many "bookkeeping" redexes. These bookkeeping redexes prevent the direct use of the cps transform as the desired translation. To overcome this problem, a modified cps transform $\overline{\overline{M}}$ is defined that contracts many of the bookkeeping redexes, that is, $\overline{M} \to_\beta^* \overline{\overline{M}}$. This modified cps transform will serve as the translation described above.

For the purposes of this proof the operator $\mathcal{A}$ will be taken as primitive and the evaluation rules of IS$_t$ will include the rule

$$\mathcal{C}(\lambda k.E[\mathcal{A}(M)]) \mapsto_{t\mathcal{A}} \mathcal{C}(\lambda k.M). \qquad (\mapsto_{t\mathcal{A}})$$

Clearly, there is no loss of generality in this assumption.

Define $\Psi(x^\alpha) = x^{\alpha^*}$, and $\Psi(\lambda x^\alpha.M) = \lambda x^{\alpha^*}.\overline{\overline{M}}$. Define

$$\overline{\overline{M}} \stackrel{\text{def}}{=} \lambda k.(M : k),$$

for k not free in M. Given a term M of type α, and a value V of type $\alpha^* \to o$, define the term $M : V$ of type o, by induction on M (it is assumed that types are chosen appropriately and that new variables are chosen to avoid capture):

1. $V_1 : V_0 = V_0\Psi(V_1)$
2. $V_1V_2 : V_0 = \Psi(V_1)\Psi(V_2)V_0$
3. $V_1N : V_0 = N : \lambda n.\Psi(V_1)nV_0$
4. $MV_1 : V_0 = M : \lambda m.m\Psi(V_1)V_0$
5. $MN : V_0 = M : (\lambda m.N : (\lambda n.mnV_0))$
6. $\mathcal{A}(M) : V_0 = M : \lambda x.\mathcal{A}(x)$
7. $\mathcal{C}(M) : V_0 = M : \lambda m.m(\lambda z.\lambda k.V_0z)\lambda x.\mathcal{A}(x)$
8. $\#\mathcal{C}(\lambda j.M) : V_0 = (M : \lambda x.\mathcal{A}(x))[(\lambda z.\lambda k.V_0z)/j]$

The special symbol $\#$ will be used to mark the top-level of a term. This definition was based on Plotkin's definition of $M : V$ in [14]. However, the $M : V$ defined here reduces more redexes and is extended to the language of IS.

The relation $\to_\beta$ denotes the usual notion of β reduction, while $\to_\beta^+$ and $\to_\beta^*$ denote the transitive, and transitive, reflexive closures, respectively, of $\to_\beta$.

Lemma 7 *For all M, $\overline{M} \to_\beta^* \overline{\overline{M}}$.*

Therefore, if M has type α, then $\overline{\overline{M}}$ has type $(\alpha^* \to o) \to o$. The following lemma states that every $\mapsto_t$ evaluation step from a term M induces zero or more $\to_\beta$ steps on the term $\overline{\overline{\#M}}$.

Lemma 8 *1. If $M_0 \mapsto_{t\beta_v} M_1$, then $\overline{\overline{\#M_0}} \to_\beta^+ \overline{\overline{\#M_1}}$.*

2. If $M_0 \mapsto_{t\mathcal{C}} M_1$, then $\overline{\overline{\#M_0}} \to_\beta^* \overline{\overline{\#M_1}}$.

3. If $M_0 \mapsto_{t\mathcal{A}} M_1$, then $\overline{\overline{\#M_0}} = \overline{\overline{\#M_1}}$.

The proof of this lemma will require the following lemmas, which are stated without proof.

Lemma 9 *For all evaluation contexts E, terms M, and values V*

$$E[M] : V = M : V^E$$

where V^E is defined by induction on E as

1. $V^{[\,]} = V$,

2. $V^{E_1N} = (\lambda m.N : (\lambda n.mnV))^{E_1}$,

3. $V^{E_1V'} = (\lambda m.m\Psi(V')V)^{E_1}$,

4. $V^{V'E_1} = (\lambda n.\Psi(V')nV)^{E_1}$.

Lemma 10 *For all M, values V,*

$$\overline{\overline{M}}[\Psi(V)/x] = \overline{\overline{M[V/x]}}.$$

[4]The author has not been able to find this embedding mentioned in the literature of proof theory.

The proof of Lemma 8 uses the abbreviations

$$\begin{array}{lcl} A & \overset{\text{def}}{=} & \lambda x.\mathcal{A}(x), \\ M^{\circ} & \overset{\text{def}}{=} & \lambda k_0.M[J(k_0)/k], \\ J(V) & \overset{\text{def}}{=} & \lambda z.\lambda d.Vz \\ & & (z, d \text{ not free in } V). \end{array}$$

Proof of Lemma 8. For the first part of the lemma, suppose

$$\begin{array}{lcl} M_0 & = & \mathcal{C}(\lambda k.E[(\lambda x.M)V]), \\ M_1 & = & \mathcal{C}(\lambda k.E[M[V/x]]). \end{array}$$

Looking at the left-hand side, we have

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_0}}} & = & (E[(\lambda x.M)V] : A)^{\circ} \\ & = & ((\lambda x.M)V : A^E)^{\circ} \\ & = & ((\lambda x.\overline{\overline{M}})\Psi(V)A^E)^{\circ} \\ & \rightarrow_{\beta} & (\overline{\overline{M}}[\Psi(V)/x]A^E)^{\circ} \\ & = & (\overline{\overline{M[V/x]}}A^E)^{\circ} \\ & = & ((\lambda k_1.M[V/x] : k_1)A^E)^{\circ} \\ & \rightarrow_{\beta} & (M[V/x] : A^E)^{\circ}. \end{array}$$

Now, turning to the right-hand side,

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_1}}} & = & (E[M[V/x]] : A)^{\circ} \\ & = & (M[V/x] : A^E)^{\circ}, \end{array}$$

which is equal to the left-hand side. For the second part of the lemma, suppose

$$\begin{array}{lcl} M_0 & = & \mathcal{C}(\lambda k.E[\mathcal{C}(N)]), \\ M_1 & = & \mathcal{C}(\lambda k.N\lambda z.\mathcal{A}(E[z])). \end{array}$$

First, note that

$$\begin{array}{lcl} \Psi(\lambda z.\mathcal{A}(E[z])) & = & \lambda z.\overline{\overline{\overline{\mathcal{A}(E[z])}}} \\ & = & \lambda z.\lambda d.(\mathcal{A}(E[z]) : d) \\ & = & \lambda z.\lambda d.(E[z] : A) \\ & = & \lambda z.\lambda d.(z : A^E) \\ & = & \lambda z.\lambda d.A^E z \\ & = & J(A^E) \end{array}$$

Looking at the left-hand side, we have

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_0}}} & = & (E[\mathcal{C}(N)] : A)^{\circ} \\ & = & (\mathcal{C}(N) : A^E)^{\circ} \\ & = & (N : \lambda m.mJ(A^E)A)^{\circ}. \end{array}$$

Turning to the right-hand side, there are two cases to consider. Suppose N is not a value, then

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_1}}} & = & (N\lambda z.\mathcal{A}(E[z]) : A)^{\circ} \\ & = & (N : \lambda m.m\Psi(\lambda z.\mathcal{A}(E[z]))A)^{\circ} \\ & = & (N : \lambda m.mJ(A^E)A)^{\circ}, \end{array}$$

and the left- and right-hand sides are equal. Suppose, on the other hand, that N is a value. Looking at the left-hand side, we have

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_0}}} & = & (N : \lambda m.mJ(A^E)A)^{\circ} \\ & = & ((\lambda m.mJ(A^E)A)\Psi(N))^{\circ} \\ & \rightarrow_{\beta} & (\Psi(N)J(A^E)A)^{\circ}, \end{array}$$

while on the right we have

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_1}}} & = & (N\lambda z.\mathcal{A}(E[z]) : A)^{\circ} \\ & = & (\Psi(N)\Psi(\lambda z.\mathcal{A}(E[z]))A)^{\circ} \\ & = & (\Psi(N)J(A^E)A)^{\circ}, \end{array}$$

which is equal to the left-hand side. Finally, for the third part of the lemma, suppose

$$\begin{array}{lcl} M_0 & = & \mathcal{C}(\lambda k.E[\mathcal{A}(N)]), \\ M_1 & = & \mathcal{C}(\lambda k.N). \end{array}$$

On the left we have

$$\begin{array}{lcl} \overline{\overline{\overline{\#M_0}}} & = & (E[\mathcal{A}(N)])^{\circ} \\ & = & (\mathcal{A}(N) : A^E)^{\circ} \\ & = & (N : A)^{\circ}, \end{array}$$

which is equal to $\overline{\overline{\overline{\#M_1}}}$. □

Lemma 11 *All sequences of $\longmapsto_{\mathcal{C}}$ steps are finite.*

Proof. Any sequence of $\longmapsto_{\mathcal{C}}$ steps must have the form

$$\begin{array}{lllll} M_0 & \propto & E_1[\mathcal{C}(M_1)] & \longmapsto_{\mathcal{C}} & M_1V_1 \\ & \propto & E_2[\mathcal{C}(M_2)]V_1 & \longmapsto_{\mathcal{C}} & M_2V_2 \\ \cdots & \cdots & & \cdots & \cdots \\ & \propto & E_i[\mathcal{C}(M_i)]V_{i-1} & \longmapsto_{\mathcal{C}} & M_iV_i \\ \cdots & \cdots & & \cdots & \cdots \end{array}$$

for some sequence of values V_1, V_2, $\cdots$, and some sequence of terms M_1, M_2, $\cdots$. This sequence must be finite since each M_{i+1} is a proper subterm of M_i and all terms have finite depth. □

By essentially the same argument we can prove the following lemma.

Lemma 12 *All evaluation sequences composed only of applications of the $\longmapsto_{t\mathcal{C}}$ and $\longmapsto_{t\mathcal{A}}$ rules are finite.*

We can now prove the main result of this section.
Proof of Theorem 6. Let M be a typed IS term of type α. Suppose there is an infinite evaluation sequence

$$\mathcal{C}\lambda k^{\neg\alpha}.M_0 \longmapsto_t \mathcal{C}\lambda k^{\neg\alpha}.M_1 \longmapsto_t \cdots$$

where $M_0 = kM$. Let $N_i = \mathcal{C}\lambda k.M_i$ and $Q_i = \overline{\overline{\overline{\#N_i}}}$. Then, by Lemma 8,

$$Q_0 \rightarrow_{\beta}^{*} Q_1 \rightarrow_{\beta}^{*} \cdots$$

where $Q_i = Q_{i+1} = \cdots = Q_{i+j}$ is possible only when the evaluation subsequence from Q_i to Q_{i+j} is composed only of $\mapsto_{t\mathcal{C}}$ and $\mapsto_{t\mathcal{A}}$ steps. Since each such subsequence is finite by Lemma 12, it must be possible to find an infinite subsequence

$$Q_0 \rightarrow^+_\beta Q'_1 \rightarrow^+_\beta Q'_2 \rightarrow^+_\beta \cdots$$

However, since Q_0 is well-typed (of type $(\alpha^* \rightarrow o) \rightarrow o$), this contradicts the well-known fact that simply typed λ-terms are strongly normalizing. Therefore, there cannot exist an infinite evaluation sequence starting from M. □

8 Conclusion

This paper has shown that a formulae-as-typed correspondence can be defined between classical propositional logic and a typed Idealized Scheme containing a control operator similar to Scheme's `call/cc`. It should be noted, however, that the paper merely presents a *formal* correspondence between classical logic and Idealized Scheme. At this point there still remains the question: Why should there be any correspondence at all? Whether or not there is a "deeper reason" underlying the correspondence is unclear at this time.

[Note: Shortly before the publication deadline for this conference the work of Andrzej Filinski [6,5] was brought to my attention. His work may provide a "deeper reason," for the correspondence described in this paper. However, due to the lack of time, I have been unable to investigate this thoroughly. Filinki defines the Symmetric Lambda Calculus (SLC), which gives a symmetric treatment of values and continuations. He then develops a categorical model of this language in which values and continuations are dual notions. Classical types for control operators seem to arise naturally in this setting.]

9 Acknowledgments

I'm indebted to Matthias Felleisen for introducing me to `call/cc`, for spending many hours patiently explaining his work in this area, and for his comments on drafts of this paper. I would like to thank Bob Harper for his comments on drafts of this paper and for bringing the work of Andrzej Filinski to my attention.

References

[1] M. Felleisen. *The calculi of λ_v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages.* PhD thesis, Indiana University, 1987. Technical Report No. 226.

[2] M. Felleisen and D. Friedman. Control operators, the secd-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141, North-Holland, 1986.

[3] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 131–141, IEEE, 1986.

[4] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[5] A. Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Summer Conference on Category Theory and Computer Science, Manchester, UK*, Springer-Verlag, 1989. to appear in the Lecture Notes in Computer Science.

[6] A. Filinski. *Declarative Continuations and Categorical Duality.* Master's thesis, University of Copenhagen, Copenhagen, Denmark, August 1989. DIKU Report 89/11, Computer Science Department.

[7] M. J. Fischer. Lambda calculus schemata. In *Proc. ACM Conference on Proving Assertions About Programs*, pages 104–109, 1972. SIGPLAN Notices 7.1.

[8] J. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Volume 7 of *Cambridge Tracts in Computer Science*, Cambridge University Press, 1989.

[9] R. J. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus. London Mathematical Society Student Texts*, Cambridge University Press, 1986.

[10] W. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490, Academic Press, NY, 1980.

[11] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964.

[12] P. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[13] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In R. Parikh, editor, *Logics of Programs*, pages 219–224, Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 193.

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

[15] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, 1965.

[16] J. Rees and W. Clinger. The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[17] G. L. Steele. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1984.

[18] S. Stenlund. *Combinators, Lambda-Terms and Proof Theory*. D. Reidel, Dordrecht, Holland, 1972.