# Chapter 5

# On the Orthogonality of Assignments and Procedures in ALGOL

*Matthias Felleisen and Stephen Weeks*

According to folklore, ALGOL is an "orthogonal" extension of a simple imperative programming language with a call-by-name functional language. The former contains assignments, branching constructs, and compound statements; the latter is based on the typed $\lambda$-calculus. To formalize ALGOL's notion of "orthogonality", we define an extended $\lambda$-calculus that models ALGOL. The calculus includes the *full* $\beta$-reduction rule and reduction rules for assignment statements and commands. It has the usual Church-Rosser property and its recursion-free subset is strongly normalizing.

In support of the "orthogonality" claim, we show that the proofs of the Church-Rosser and Strong-Normalization theorems are combinations of separate theorems for each sub-language. In addition, the calculus satisfies a Postponement Theorem which operationally formalizes the idea that the evaluation of an ALGOL program has two distinct phases which correspond to the two sub-languages. The first phase corresponds to an unrolling of the program according to the usual $\beta$ and fixpoint reductions. The result of this phase is essentially an imperative program. The second phase corresponds to the execution of an imperative program on an ordinary stack machine.

## Contents

1 Introduction 102
2 An Idealized Dialect 102
2.1 Syntax and Informal Semantics 103
2.2 Semantics and Calculus 105
2.3 Characteristics of *eval* 107
3 Postponement 109
4 Strong Normalization 113
5 Extensions and Alternatives 115
Acknowledgements 116
References 116
A Church-Rosser Theorem for *ia* 118
B Uniform Evaluation 120
C Postponement 121
D Strong Normalization 123

Preliminary version appeared in *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 57–70, Charleston, South Carolina, 1993. ACM, New York.

## 1 Introduction

According to folklore, ALGOL 60 "orthogonally" extends a simple imperative programming language with a typed λ-calculus. The underlying imperative language consists of a standard collection of constructs: assignment statements, branching statements, loops, and statement sequences. The typed λ-calculus adds recursive higher-order procedures, which provide the power to abstract over algorithms in the command language. In his most recent description of FORSYTHE, a modern variant of ALGOL, Reynolds expounds this view, which he says is

> implicit in ALGOL 60, underlies FORSYTHE, and distinguishes it from such languages as ALGOL 68, SCHEME, and ML: the programming language is a typed lambda calculus with a primitive type **comm**, such that terms of this type, when reduced to normal form, are programs in the simple imperative language. [18, p. 3]

Moreover, he reiterates the common belief that, as a result of this design, a FORSYTHE program

> is executed in two phases. First the program is reduced to normal form. (In ALGOL jargon, the copy rule is repeatedly applied to eliminate procedure calls.) Then the resulting simple imperative program is executed. [18, p. 3]

Given the recent interest in integrating variations of assignment statements into functional programming languages in a controlled manner and in the design of simple logics for such mixed languages [2, 8, 9, 11, 12, 15, 21], these folklore claims about ALGOL's design clearly deserve a rigorous analysis. On one hand, such an analysis enhances our understanding of the role of "orthogonality" in ALGOL.[1] On the other hand, the results improve our understanding of the simple equational logics for mixed functional-imperative programming languages and of their abstract implementation.

The next section introduces the syntax, calculus, and semantics of a small ALGOL-like language. The calculus includes the *full* $\beta$-rule, despite the presence of assignments. Section 2.3 addresses the basic consistency results for the calculus and the derived semantics, i.e., Church-Rosser, Standardization, and Strong Normalization for the recursion-free subset. Section 3 contains the Postponement Theorem, which shows how the evaluation of a program can indeed be separated into a functional phase followed by an imperative phase. Section 4 contains the proof that Postponement combined with Strong Normalization for each sub-calculus entails Strong Normalization for the complete calculus. The last section lists the shortcomings of our ALGOL dialect and includes a brief discussion of related work.

## 2 An Idealized Dialect

Our version of IDEALIZED ALGOL (IA) is an idealized variant of ALGOL 60, similar to, but significantly simpler than, Reynolds's FORSYTHE [18, 16, 17]. Specifically, the type system is that of the simply-typed λ-calculus; the imperative

[1] This goal is comparable to the attempts of Søndergard and Sestoft [19] to clarify the often misused terminology of "referential transparency" and of Felleisen [5] to clarify the concept of "expressive power of programming languages."

sub-language of IA contains only arithmetic primitives; and the language lacks facilities for handling exceptions or dealing with complex data. Thus, it is simple enough to permit an easily comprehensible semantics, yet complete enough to permit a generalization of our results to other members of the ALGOL-family.

Based on the syntactic definition in the first subsection, we specify the semantics of the language in the second subsection via a combination of two reduction systems. One system describes the command component, the other one is a simply-typed $\lambda$-calculus. The union of the two basic reduction relations generates the calculus for the entire language. We then show in the final subsection that this calculus provides an adequate reasoning system for the language; i.e., it is consistent and strong enough to define a well-behaved evaluation function.

### 2.1 Syntax and Informal Semantics

The following definition specifies the abstract (context-free) syntax of IA's commands and expressions:

| | | | |
|---|---|---|---|
| $M$ | ::= | $\ulcorner n \urcorner \mid op \mid x \mid (M\ M)$ | arithmetic expressions |
| | | $\mid$ **skip** $\mid$ **if** $M\ M\ M \mid$ **begin** $M\ M$ | nop, branching, sequencing |
| | | $\mid$ **new**$(x, M).M$ | block |
| | | $\mid\ !M \mid M := M$ | dereferencing, assignment |
| | | $\mid \lambda x.M \mid$ (**rec** $M$) | (recursive) procedures |

where $n \in \mathbb{Z}$, $op \in \{+, -, *\}$, and $x \in \mathit{Vars}$, a denumerable set of variables.

The first part of the set of terms (above the line) constitutes the syntax of the imperative sub-language; the second part (below the line) is the $\lambda$-calculus extension, including a **rec**-construct for declaring recursive objects (procedures and the "diverging integer"). Both $\lambda x.M$ and **new**$(x, N).M$ *bind* $x$ in $M$; no other construct binds a variable in a term. A variable occurs *free* in a term if it is not bound by a surrounding binding construct. A *context* is a term with a single "hole", [ ], in the place of a sub-term; $C[\ ]$ denotes a context. The notation $C[M]$ refers to the result of "*filling*" the hole of the context $C[\ ]$ with term $M$, possibly capturing some free variables of $M$.

**Convention 1.** We adopt a number of Barendregt's [1] $\lambda$-calculus conventions for IA. Specifically, we identify terms that are equivalent modulo change of bound variables and use $M \equiv N$ to denote this equivalence. The sets of free and bound variables in distinct terms do not interfere with each other in definitions and theorems. Contexts are *not* subject to this convention. ∎

The type system filters legal from illegal terms and separates the set of terms into a sub-language of commands (of type **comm**) and expressions. It is a simple version of FORSYTHE's type system [17].

An expression in the imperative sub-language may either be of ground type ($o$) or of a first-order arithmetic function type ($o \to o$) and $o \to (o \to o)$. For example, $\ulcorner n \urcorner$ is of type $o$ and $(+\ \ulcorner 1 \urcorner)$ is of type $o \to o$. A command in the imperative sub-language is a term of type **comm**. Like ALGOL and FORSYTHE,

- Constants

$$\pi \rhd \ulcorner n \urcorner : \mathbf{int}$$
$$\pi \rhd op : \mathbf{int}\to\mathbf{int}\to\mathbf{int}$$

- Variables

$$\pi \rhd x : \pi(x),\ x \in \mathrm{dom}(\pi)$$

- Functions

$$\frac{\pi[x/\tau] \rhd M : \tau'}{\pi \rhd \lambda x.M : \tau\to\tau'}$$
$$\frac{\pi \rhd M : \tau\to\tau}{\pi \rhd (\mathbf{rec}\ M) : \tau}$$
$$\frac{\pi \rhd M : \tau'\to\tau,\ \pi \rhd N : \tau'}{\pi \rhd (M\ N) : \tau}$$

- References

$$\frac{\pi \rhd M : \mathbf{int},\ \pi[x/\mathbf{int\ ref}] \rhd N : \varsigma}{\pi \rhd \mathbf{new}(x, M).N : \varsigma}$$
$$\frac{\pi \rhd M : \mathbf{int\ ref}}{\pi \rhd\ !M : \mathbf{int}}$$
$$\frac{\pi \rhd M : \mathbf{int\ ref},\ \pi \rhd N : \mathbf{int}}{\pi \rhd M\ :=\ N : \mathbf{comm}}$$

- Simple Commands & Expressions

$$\pi \rhd \mathbf{skip} : \mathbf{comm}$$
$$\frac{\pi \rhd L : \mathbf{int},\ \pi \rhd M : o,\ \pi \rhd N : o}{\pi \rhd \mathbf{if}\ L\ N\ N : o}$$
$$\frac{\pi \rhd M : \mathbf{comm},\ \pi \rhd N : o}{\pi \rhd \mathbf{begin}\ M\ N : o}$$

Figure 1: Type Inference System for IA

IA restricts references to contain only basic data values (**int**'s), and the results of **new**-blocks are of ground type.

The extension of the imperative sub-language to IA is accomplished by relaxing the constraint on expression types. The full language admits expressions of type $\tau\to\tau$ for arbitrary types $\tau$; e.g., **int**→**int**→**comm** is a permissible type now. The restrictions on references and **new**-blocks remain.

For the formal specification of the type system, we begin with a definition of the full language of types:

$$\begin{array}{lll} \tau & ::= & o \mid \tau\to\tau \\ o & ::= & \mathbf{comm} \mid \mathbf{int} \mid \mathbf{int\ ref} \\ \varsigma & ::= & \mathbf{comm} \mid \mathbf{int} \end{array}$$

The type inference system is defined by a set of *inference rules*, whose basic components are *typings*. A typing, $\pi \rhd M : \tau$ consists of a *type assignment*, $\pi$, an expression, $M$, and a type, $\tau$. A type assignment, $\pi$, is a finite function from variables to types. We use the notation $\pi[x/\tau]$ to indicate the type assignment such that $\pi[x/\tau](x) = \tau$ and $\pi[x/\tau](y) = \pi(y)$, if $y \not\equiv x$. The typing $\pi \rhd M : \tau$ asserts that expression $M$ has type $\tau$ when the free variables in $M$ are assigned types by $\pi$. Figure 1 contains the inference rules for IA.

While the type inference system is a straightforward extension of that for the simply typed λ-calculus, two apparent restrictions deserve some comments. Both **if**-expressions and **begin**-expressions are commands as well as expressions, depending on which ground type they are assigned. In addition,

the requirement that **if**-expressions and **begin**-expressions be of type $o$ is only to simplify the theory, and may be removed in practice with the following abbreviations:

$$\begin{array}{lcl} \textbf{if}\ L\ M\ N & \stackrel{df}{\equiv} & \lambda\vec{x}.\,\textbf{if}\ L\ (M\ \vec{x})\ (N\ \vec{x}) \\ \textbf{begin}\ M\ N & \stackrel{df}{\equiv} & \lambda\vec{x}.\,\textbf{begin}\ M\ (N\ \vec{x}); \end{array}$$

in each case, $\vec{x}$ is a vector of variables of the length required to assign an appropriate ground type to the expressions $(M\ \vec{x})$ and $(N\ \vec{x})$.

The behaviour of IA's imperative sub-language is that of an assembly language for a stack-machine. Numerals are basic values; arithmetic expressions are evaluated as usual. Blocks allocate a new location (reference) with some initial value, execute their body, and finally deallocate the location. The compound and branching terms are also executed in the usual manner: the former sequences the execution of two terms, the latter compares the result of its first sub-expression to $\ulcorner 0 \urcorner$ and branches to the second if the answer is affirmative and to the third term otherwise.

The informal semantics of the procedural language is easily explained via ALGOL's copy rule [13]. A procedure replaces its parameters by its arguments (after renaming internal bound variables, which we implicitly assume according to Convention 1). A recursive definition is unrolled as far as necessary.

The sub-language of IA does not provide a looping construct but again, this omission only simplifies the analysis of the theory and is no real obstacle to programming in IA. For example, a **while** loop is a simple abbreviation, based on recursive functions:

$$\textbf{while}\ M\ do\ N \stackrel{df}{\equiv} (((\textbf{rec}\ \lambda w.\,\lambda x.\,\lambda y.\,\textbf{if}\ x\ \textbf{skip}\ \textbf{begin}\ y\ ((w\ x)\ y))\ M)\ N)$$

The procedure $\Sigma_!$

$$\begin{array}{l} \lambda j.\textbf{new}(S,\ulcorner 0\urcorner).\textbf{new}(i,j). \\ \quad (\textbf{begin}\ \textbf{while}\ (!\mathrm{i})\ \textbf{do}\ (\textbf{begin}\ S := ((+\ !S)\ !\mathrm{i}) \\ \qquad\qquad\qquad\qquad\qquad\qquad i := ((-\ !i)\ \ulcorner 1\urcorner)) \\ \qquad !S) \end{array}$$

employs a mixture of functional and imperative facilities. It passes the type inference system as an expression of type **int**→**int**, that is, as a procedure from integers to integers. $(\Sigma_!\ j)$ computes the sum from $\ulcorner 0 \urcorner$ to $j$ in an imperative fashion. Here is an equivalent functional version:

$$\Sigma_\lambda \stackrel{df}{\equiv} (\textbf{rec}\ \lambda sum.\,\lambda i.\,\textbf{if}\ i\ \ulcorner 0\urcorner\ ((+\ i)\ (sum\ ((-\ i)\ \ulcorner 1\urcorner))))$$

### 2.2 Semantics and Calculus

To formalize IA's semantics, we define an equational calculus.[2] The definition of the basic notion of reduction for IA relies heavily on our experience with extended λ-calculi for untyped languages with assignment statements, specifically, the definition of the $\lambda_n$-R-calculus for a call-by-name language and that of the $\lambda_v$-B-calculus for a language with first-class reference cells [2, 8].

[2] In contrast, Lent [10] defines an operational semantics of an IA-like language based on so-called structural operational semantics. His major concern is an "adequacy theorem" for his language with respect to a categorical model.

| | | | |
|---|---|---|---|
| $(\delta)$ | $((op\ \ulcorner n\urcorner)\ \ulcorner m\urcorner)$ | $\longrightarrow$ | $\ulcorner n\ op\ m\urcorner$ |
| (**pop**) | $\mathbf{new}(x, \ulcorner n\urcorner).\, v$ | $\longrightarrow$ | $v$, $v$ is a numeral or **skip** |
| $(D)$ | $\mathbf{new}(x, \ulcorner n\urcorner).\, E[!x]$ | $\longrightarrow$ | $\mathbf{new}(x, \ulcorner n\urcorner).\, E[\ulcorner n\urcorner], x \notin \mathrm{TR}(E)$ |
| $(\sigma)$ | $\mathbf{new}(x, \ulcorner n\urcorner).\, E[x := \ulcorner m\urcorner]$ | $\longrightarrow$ | $\mathbf{new}(x, \ulcorner m\urcorner).\, E[\mathbf{skip}], x \notin \mathrm{TR}(E)$ |
| (**T**) | $\mathbf{if}\ \ulcorner 0\urcorner\ M\ N$ | $\longrightarrow$ | $M$ |
| (**F**) | $\mathbf{if}\ \ulcorner n+1\urcorner\ M\ N$ | $\longrightarrow$ | $N$ |
| (**B**) | $\mathbf{begin\ skip}\ M$ | $\longrightarrow$ | $M$ |
| $(\beta)$ | $((\lambda x.\, M)\ N)$ | $\longrightarrow$ | $[N/x]M$ |
| (**fix**) | $(\mathbf{rec}\ M)$ | $\longrightarrow$ | $(M\ (\mathbf{rec}\ M))$ |

Figure 2: Basic Notions of Reduction

**Convention 2.** We adopt Barendregt's [1] conventions about reduction relations and calculi. A *notion of reduction*, $\mathbf{r}$, denotes a binary relation on the set of phrases. The relation $\longrightarrow_{\mathbf{r}}$, the compatible closure of $\mathbf{r}$, is defined by:

$$(M, N) \in \mathbf{r} \Rightarrow C[M] \longrightarrow_{\mathbf{r}} C[N] \text{ for all contexts } C.$$

The relation $\twoheadrightarrow_{\mathbf{r}}^{*}$ is the reflexive and transitive closure of $\twoheadrightarrow_{\mathbf{r}}$. The relation $=_{\mathbf{r}}$ is the equivalence relation generated by $\twoheadrightarrow_{\mathbf{r}}$. Given two notions of reduction $\mathbf{r}_1$ and $\mathbf{r}_2$, $\mathbf{r}_1\mathbf{r}_2$ denotes $\mathbf{r}_1 \cup \mathbf{r}_2$. ∎

In order to describe the rules for assignment, we must first define a set of special contexts, *evaluation contexts*, and the corresponding set of trapped references, $\mathrm{TR}(E)$:

| $E$ ::= | $\mathrm{TR}(E)$ | $E$ ::= | $\mathrm{TR}(E)$ |
|---|---|---|---|
| $\mid [\ ]$ | $\varnothing$ | $\mid\ !E$ | $\mathrm{TR}(E)$ |
| $\mid ((op\ E)\ M)$ | $\mathrm{TR}(E)$ | $\mid\ E := M$ | $\mathrm{TR}(E)$ |
| $\mid ((op\ \ulcorner n\urcorner)\ E)$ | $\mathrm{TR}(E)$ | $\mid\ x := E$ | $\mathrm{TR}(E)$ |
| $\mid \mathbf{new}(x, E).\, M$ | $\mathrm{TR}(E)$ | $\mid\ \mathbf{if}\ E\ M\ M$ | $\mathrm{TR}(E)$ |
| $\mid \mathbf{new}(x, \ulcorner n\urcorner).\, E$ | $\{x\} \cup \mathrm{TR}(E)$ | $\mid\ \mathbf{begin}\ E\ M$ | $\mathrm{TR}(E)$ |

The hole in an evaluation context indicates the sub-expression whose assignment commands and dereferencing expressions may be executed next. It is an easy induction to show that a term has a unique partitioning into an evaluation context and an assignment command or dereferencing expression, unless it is a numeral or a **skip**. This fact guarantees the well-behavedness of the imperative part of the calculus; we call this the *unique-partition property*.

Figure 2 defines the basic notions of reduction for IA. The first part (above the line) characterizes the behaviour of the imperative sub-language and the second part (below the line) is the usual set of reductions for the simply-typed, call-by-name $\lambda$-calculus with a recursion construct.

The rules capture the informal semantics in an intuitive manner. Sequencing of the **begin**-expression is enforced by the structure of evaluation contexts

and the **B** rule. Together, these only allow non-local assignments in the second subexpression of a **begin**-expression to take place after the first subexpression has been completely evaluated to a **skip** and removed by the **B** rule. Despite the presence of assignments, function applications satisfy the *full* $\beta$-axiom. Potential conflicts between assignments to and dereferences of the same references are eliminated by the use of evaluation contexts due to the above-mentioned unique-partition property for phrases. In contrast, assignments to distinct references may be reduced in parallel. For example, during the evaluation of

$$((+\ (\Sigma_!\ \ulcorner 10 \urcorner))\ (\Sigma_!\ \ulcorner 5 \urcorner))\ ,$$

the assignments in the two distinct procedure calls of $\Sigma_!$ can be reduced independently, which implies that a machine could execute them in parallel.[3]

Based on the primitive notions of reduction, we define three compound term relations. The first, *ia*, is intended to describe evaluation in the full language IA. The second, $\beta$**fix**, corresponds to the axioms describing the functional part of the language. The third, $\rho$, corresponds to the axioms describing the command component of the language:

$$\begin{array}{lcl} \mathit{ia} & \stackrel{df}{\equiv} & \beta\mathbf{fix} \cup \rho \\ \beta\mathbf{fix} & \stackrel{df}{\equiv} & \beta \cup \mathbf{fix} \\ \rho & \stackrel{df}{\equiv} & \delta \cup \mathbf{pop} \cup D \cup \sigma \cup \mathbf{T} \cup \mathbf{F} \cup \mathbf{B} \end{array}$$

To emphasize that the above is a calculus, we sometimes write $\mathbf{r} \vdash M = N$ when $M =_{\mathbf{r}} N$ where $\mathbf{r}$ is either *ia*, $\beta$**fix**, or $\rho$.

Given the calculus, we can specify the semantics of IA. An IA program is a closed integer expression. Its result is any numeral to which it is provably equal.

**Definition 1** A *program* $M$ is a closed expression of type **int**; i.e.,

$$M \in \mathit{Programs} \text{ if and only if } \varnothing \rhd M : \mathbf{int}.$$

The *evaluator* is a partial function from programs to numerals:

$$\mathit{eval}: \mathit{Programs} \rightharpoonup \mathit{Numerals}.$$

If $M$ is a program, then $\mathit{eval}(M) = \ulcorner n \urcorner$ if and only if $\mathit{ia} \vdash M = \ulcorner n \urcorner$. We write $\mathit{eval}_\rho(M) = \ulcorner n \urcorner$ when $\rho \vdash M = \ulcorner n \urcorner$.

### 2.3 Characteristics of *eval*

To show that the language IA is a deterministic, well-behaved language like ALGOL, we need to prove that the evaluator is indeed a function, and that this function is only undefined if all reductions starting from the program are infinite. The easiest way to prove these results is via a Church-Rosser theorem and a subject reduction theorem for types. Both results hold for the two sublanguages and carry over to IA in a uniform manner.

We begin by proving that *eval* is a well-defined function.

[3] Not all expressions whose effects do not interfere can be reduced independently in the current calculus.

**Theorem 2** *Let $M$ be a program, let $n, m \in \mathbb{Z}$.*

1. *If $eval(M) = \ulcorner n \urcorner$ and $eval(M) = \ulcorner m \urcorner$ then $m = n$.*
2. *$eval(M) = \ulcorner n \urcorner$ if and only if $M \longrightarrow_{ia}^{*} \ulcorner n \urcorner$.*

**Proof:** Both parts are immediate consequences of Theorem 3. ∎

**Theorem 3 (Church-Rosser Theorem)** *If $L \longrightarrow_{ia}^{*} M$ and $L \longrightarrow_{ia}^{*} N$, then there exists $K$ such that $M \longrightarrow_{ia}^{*} K$ and $N \longrightarrow_{ia}^{*} K$.*

**Proof:** The proof consists of three lemmas. The first two establish the Church-Rosser property for each subsystem, i.e., $\rho$ and $\beta$**fix**, separately. The third lemma shows that the subsystems merge smoothly.

1. The notion of reduction $\beta$**fix** is Church-Rosser. This fact immediately follows from the Church-Rosser Theorem for simply-typed $\lambda$-calculus (with **rec**). The new syntax does not interfere with the proof.
2. The reflexive-compatible closure of $\rho$ satisfies the diamond property directly. This is shown by careful case analysis: see Appendix A, Lemma 19 for details. Hence, by a diagram chase, the reflexive and transitive closure of $\longrightarrow_{\rho}$ satisfies the diamond property, and $\rho$ is Church-Rosser.
3. The two reductions generated by $\beta$**fix** and $\rho$ commute. This is shown by defining a parallel reduction relation $\xrightarrow[1]{}_{\rho}$, and showing that $\longrightarrow_{\beta\mathbf{fix}}$ and $\xrightarrow[1]{}_{\rho}$ commute: see Lemma 22 for details.

The Church-Rosser Theorem for *ia* follows directly from the three lemmas by the Hindley-Rosen Lemma [1, p. 64]. ∎

The second important property of the evaluator for IA is that it never gets "stuck". That is, we can show that every program either goes into an infinite loop or terminates giving a numeral. This result is the subject of the following theorem.[4]

**Theorem 4 (Uniform Evaluation)** *For all programs $M$, either $M \longrightarrow_{ia}^{*} \ulcorner n \urcorner$, for some $n \in \mathbb{Z}$ or, for all $N$ such that $M \longrightarrow_{ia}^{*} N$, there exists $N'$ such that $N \longrightarrow_{ia} N'$.*

**Proof:** The result follows from two lemmas. The first (Lemma 5) shows that reductions preserve types, and hence that a program always reduces to a program. The proof of the second (Lemma 6) provides a characterization of certain normal forms, and in particular shows that a program in normal form must be a numeral. ∎

**Lemma 5** *If $M \longrightarrow_{ia} N$ and $\pi \rhd M : \tau$, then $\pi \rhd N : \tau$.*

**Proof:** By induction on the structure of $M$: The base cases are vacuously true. The inductive cases require an easy application of the inductive hypothesis or a careful inspection of the type of the contractum of a redex. See Appendix B, Lemma 25 for details. ∎

[4] If IA contained integer division (or other partial primitive functions), then the calculus *ia* would have to include error values and reductions for error values. The Uniform-Evaluation Theorem would have to state that a program either reduces to a numeral or an error value.

**Lemma 6** *If $P$ is a program in normal form (that is to say, there is no $P'$ such that $P \longrightarrow_{ia} P'$), then $P = \ulcorner n \urcorner$ for some $n \in \mathbb{Z}$.*

**Proof:** We prove the following more general lemma by induction on the structure of an arbitrary expression $N$:

$$\text{if } \{(x_1, \textbf{int ref}), \ldots, (x_n, \textbf{int ref})\} \rhd N : \tau,$$

and $N$ is a normal form, then $N \in t_1$, the language described by the following grammar:

$$\begin{array}{lll} t_1 & ::= & \ulcorner n \urcorner \mid op \mid (op\ \ulcorner n \urcorner) \mid (op\ t_2) \mid x_i \mid \lambda y.M \mid \textbf{skip} \mid t_2 \\ t_2 & ::= & E[!x_i] \mid E[x_i := \ulcorner n \urcorner], \text{ where } x_i \notin \mathrm{TR}(E) \end{array}$$

The specific result follow by noting that a program $M$ must satisfy $\varnothing \rhd M : \textbf{int}$. See Lemma 27 for details. ∎

Theorems 2 and 4, together with Theorem 3, show that the calculus defines a well-behaved interpreter and that it is sound with respect to program equivalences [2]. Also, the proofs of the preceding theorems and lemmas show that the functional and imperative sub-languages are independent and that meta-properties follow from modularized proofs. In the next section we will prove that the evaluation itself can proceed in a highly modular fashion.

## 3 Postponement

The restrictions on ALGOL's type system are motivated by the desire to execute programs in two phases. The first phase eliminates procedures and their uses by reducing programs with $\beta\textbf{fix}$ sufficiently far. The result is a mixed program whose functional components are irrelevant for the rest of the evaluation. The second phase executes the imperative program according to the $\rho$-rules, which can be implemented with a regular stack machine. The following theorem makes this idea precise.

**Theorem 7** *Let $M$ be a program, let $n \in \mathbb{Z}$. Then,*

$$\mathit{eval}(M) = \ulcorner n \urcorner \quad \mathit{iff} \quad \mathit{for\ some\ program}\ N, M \longrightarrow^*_{\beta\textbf{fix}} N \longrightarrow^*_{\rho} \ulcorner n \urcorner$$

**Proof:** The direction from right to left is trivial, the other direction is difficult. We proceed as follows. The first step is to define a parallel reduction relation, $\xrightarrow[1]{}_\rho$, that contains $\longrightarrow_\rho$ and is contained in $\longrightarrow^*_\rho$ (see Definition 21). This follows the method of Tait/Løf for the Church-Rosser Theorem. The second step is to replace all $\longrightarrow_\rho$ steps with $\xrightarrow[1]{}_\rho$ steps in the reduction $M \longrightarrow^*_{ia} \ulcorner n \urcorner$, which must exist since $\mathit{eval}(M) = \ulcorner n \urcorner$. The final step is to prove that we can push all $\longrightarrow_{\beta\textbf{fix}}$ to the left and all $\xrightarrow[1]{}_\rho$ to the right (see Appendix C, Lemma 29 for details). The proof method is inspired by Plotkin's proof of standardization for the untyped λ-calculus [14, p. 140]. ∎

According to this theorem, we may view compilation as reduction in the functional fragment and execution as reduction in the imperative fragment. However, Reynolds points out that recursion presents a major obstacle because

> the reduction phase may go on forever, producing an infinite "head-normal" term. Nevertheless, such an infinite term can still be viewed as a simple imperative program; operationally, one simply implements the two phases as coroutines. [18, p. 3]

Put differently, a compiler cannot know in advance how far to unroll a recursive program. Hence, the compiler must unroll it all the way, which means that the result is an infinite term. The important point is that although this infinite term is in (an extended version of) the imperative language, it can still be executed in the usual imperative reduction system.

In order to formalize these notions, we use the following strategy. First, we extend the language of IA with an additional constant, $\Omega$, denoting observable nontermination and define the function $[\![\cdot]\!]_i$ from expressions in IA to the extended language. The translation unrolls all recursive functions $i$ times. Second, we define **W**, a purely imperative sublanguage of extended IA that precisely characterizes the $\beta$-normal-forms of unrolled programs. This imperative sublanguage may be viewed as the target language of the compiler. Finally, we prove a new Postponement theorem for the extended language.

We start with the definition of the extended expression language and the "unrolling" translation.

**Definition 8** The $i$'th *unrolling* of an IA expression $M$, notation: $[\![M]\!]_i$, is defined by induction on the structure of M:

$$[\![(\mathbf{rec}\ M)]\!]_i = (\underbrace{[\![M]\!]_i(\cdots([\![M]\!]_i}_{i \text{ times}}\ \Omega)))$$

$$\begin{aligned}
[\![b]\!]_i &= b & [\![\mathbf{skip}]\!]_i &= \mathbf{skip}\\
[\![op]\!]_i &= op & [\![\mathbf{if}\ M_1\ M_2\ M_3]\!]_i &= \mathbf{if}\ [\![M_1]\!]_i\ [\![M_2]\!]_i\ [\![M_3]\!]_i\\
[\![x]\!]_i &= x & [\![\mathbf{begin}\ M_1\ M_2]\!]_i &= \mathbf{begin}\ [\![M_1]\!]_i\ [\![M_2]\!]_i\\
[\![\lambda x.M]\!]_i &= \lambda x.[\![M]\!]_i & [\![\mathbf{new}(x, M_1).M_2]\!]_i &= \mathbf{new}(x, [\![M_1]\!]_i).[\![M_2]\!]_i\\
[\![(M_1\ M_2)]\!]_i &= ([\![M_1]\!]_i\ [\![M_2]\!]_i) & [\![!M]\!]_i &= ![\![M]\!]_i\\
& & [\![M_1 := M_2]\!]_i &= [\![M_1]\!]_i := [\![M_2]\!]_i
\end{aligned}$$

We refer to $i$ as the unrolling index. The constant $\Omega$ is of ground type; for higher types, it is an abbreviation for $\lambda\vec{x}.\Omega$ for an appropriate vector $\vec{x}$ of variables.

Unrolling a program $i$ times roughly corresponds to using the axiom (**fix**) $i$ times on each **rec**-expression. More precisely,

$$(\mathbf{rec}\ M) \longrightarrow_{\mathbf{fix}}^{i} \underbrace{(M(\cdots(M}_{i \text{ times}}\ (\mathbf{rec}\ M))\cdots)$$

but

$$[\![(\mathbf{rec}\ M)]\!]_i = (\underbrace{[\![M]\!]_i(\cdots([\![M]\!]_i}_{i \text{ times}}\ \Omega))).$$

To formalize the precise relationship between the two operations, we introduce an ordering, $\sqsubseteq$, on terms. It is the usual prefix-ordering for terms with respect to $\Omega$.

**Definition 9** The relation $\sqsubseteq$ is defined as follows:

1. $\Omega \sqsubseteq M$
2. $M \sqsubseteq M$
3. if $M \sqsubseteq M'$ and $C \sqsubseteq C'$ then $C[M] \sqsubseteq C'[M']$

For contexts, $C \sqsubseteq C'$ if $C[\Omega] \sqsubseteq C'[\Omega]$.

Thus, unrolling a term $L$ produces a term that is below some term $L'$ that results from $L$ via **fix**.

**Lemma 10** *For all $i \in \mathbb{N}$, $L \in IA$, there exists a term $L'$ such that*

$$[\![L]\!]_i \sqsubseteq L' \text{ and } L \longrightarrow_{\mathbf{fix}}{}^{*} L';$$

**Proof:** By induction on the structure of $L$. ∎

The target language of the "compiler" is the purely imperative sub-language, **W**, which is essentially extended IA without recursion, λ-abstraction, or arbitrary function application.

**Definition 11** The *imperative sub-language* **W** of extended IA is defined by the following grammar:

$$\begin{array}{rcl} t & ::= & \Omega \mid \ulcorner n \urcorner \mid x \mid ((op\ t)\ t) \\ & & \mid \mathbf{new}(x, t).\ t \mid !\,t \mid t := t \\ & & \mid \mathbf{skip} \mid \mathbf{if}\ t\ t\ t \mid \mathbf{begin}\ t\ t \end{array}$$

Legal phrases of **W** are those that satisfy the type inference rules of Figure 1 with the additional constraint that $\Omega$ is of one of the groundtypes ($o$).

In order to show that **W** is the proper target language, we prove that the result of $\beta$-reducing an unrolled program to $\beta$-normal-form is always in **W**.

**Lemma 12** *For all $i \in \mathbb{N}$, $L \in IA$, the $\beta$-normal-form of $[\![L]\!]_i$ is in* **W**.

**Proof:** The proof of the lemma reduces to showing that both the function and argument position of an application in a normal form expression cannot be a λ-expression. See Appendix C, Lemma 30 for details. ∎

Since it is impossible to know *a priori* how far recursive procedures must be unrolled, the "compilation" of an IA program into a **W** program must produce the set of $\beta$-normal-form's of *all* unrollings of the original program. Hence, to understand the evaluation of an IA program as the execution of a **W** program, we need to extend $eval_\rho$ (see Definition 1) to *sets* of **W** programs.

**Definition 13** Let $W$ be a set of **W** programs, i.e., a set of closed phrases of ground type. Then, $eval_\rho$ applied to this set is the pointwise extension of the original evaluator:

$$eval_\rho(W) = \{eval_\rho(w) \mid w \in W\} = \{\ulcorner n \urcorner \mid w \longrightarrow_\rho^* \ulcorner n \urcorner, w \in W\}.$$

To prove that $eval_\rho$ is equivalent to the original evaluator, we must show that both the functional and imperative reduction system respect the prefix ordering, $\sqsubseteq$. If this holds, the $\beta$-normal-forms of all finitely unrolled versions of a program clearly form a totally ordered, infinite set of **W** programs, which may be perceived as the infinite $\beta$-normal-form of an IA program. If the imperative executions of all elements of the infinite term preserve the ordering, and if one of the elements reduces to the correct final answer, then the two evaluators indeed agree. We begin by proving that the functional reductions preserve the approximation ordering.

**Lemma 14** *For all programs $M$ and $i \in \mathbb{N}$,*

$$\textit{the } \beta\textit{-normal-form of } [\![M]\!]_i \sqsubseteq \textit{the } \beta\textit{-normal-form of } [\![M]\!]_{i+1}$$

**Proof:** The result follows from Lemmas 32 and 33. The first one shows that $[\![M]\!]_i \sqsubseteq [\![M]\!]_{i+1}$. The latter shows that $\beta$ normalization respects $\sqsubseteq$. See Appendix C for details. ∎

Second, we prove that imperative reductions preserve approximations. In particular, if a **W** program terminates, then all dominating programs do.

**Lemma 15** *For all $M, N \in \mathbf{W}$, if $M \sqsubseteq N$ and $M \twoheadrightarrow_\rho {}^\ulcorner n^\urcorner$, then $N \twoheadrightarrow_\rho {}^\ulcorner n^\urcorner$.*

**Proof:** By induction on the length of the reduction $M \twoheadrightarrow_\rho {}^\ulcorner n^\urcorner$. For a single step $M \twoheadrightarrow_\rho M_1$, since $M \sqsubseteq N$, either $M \equiv M_1$ if it is an $\Omega$ reduction or the "same" redex exists in $N$. The first case is trivial. For the second case, it is simple to check that reducing the corresponding redex in $N$ produces a phrase $N_1$ such that $M_1 \sqsubseteq N_1$, hence the induction hypothesis may be applied. ∎

Finally, we state and prove a more general version of the postponement theorem that characterizes compilation as $\beta$-normalization to an infinite tree and machine execution as an evaluation of the infinite tree in the imperative fragment.

**Theorem 16** *For all IA programs $L$,*

$$\{eval(L)\} = eval_\rho(\{M \mid M \textit{ is } \beta\textit{-normal-form of } [\![L]\!]_i \textit{ for } i \in \mathbb{N}\}).$$

**Proof:** Let $W = \{M \mid M \text{ is } \beta\text{-normal-form of } [\![L]\!]_i \text{ for } i \in \mathbb{N}\}$. By Lemma 14, $W$ is a chain, i.e., a totally ordered set. Thus, the theorem reduces to the claim:

$$eval(L) = {}^\ulcorner n^\urcorner$$

if and only if there exists some $i$ and $\beta$-normal-form $M$ such that

$$[\![L]\!]_i \twoheadrightarrow_\beta^* M \twoheadrightarrow_\rho {}^\ulcorner n^\urcorner$$

If this claim is true, then, by Lemma 15, $W$ contains at most one result, which must be the result of the original program.

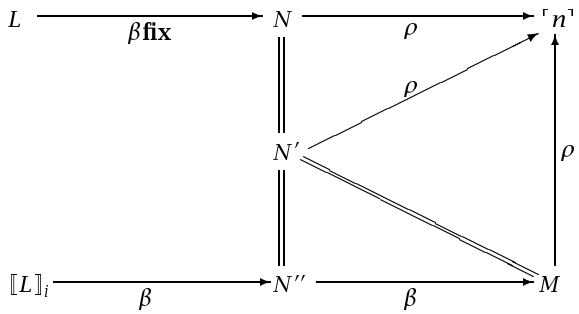
The right to left direction of the auxiliary claim is obvious. For the left to right direction, by the Postponement Theorem 7, $eval(L) = {}^\ulcorner n^\urcorner$ implies that there is a reduction:

$$L \twoheadrightarrow_{\beta\mathbf{fix}}^* N \twoheadrightarrow_\rho {}^\ulcorner n^\urcorner.$$

Clearly, no **rec**-expression or $\lambda$-expression in $N$ is relevant to the reduction from $N$ to ${}^\ulcorner n^\urcorner$. Hence we may replace all **rec**-expressions and $\lambda$-expressions in

$N$ by $\Omega$ to produce a term $N'$ such that $N' \sqsubseteq N$ and $N' \longrightarrow_\rho^* \ulcorner n \urcorner$. Next, let $i$ be the number of **fix** steps in the reduction from $L$ to $N$. Directly corresponding to the sequence of $\beta$ reductions in $L \longrightarrow_{\beta\mathbf{fix}}^* N$ is a sequence of $\beta$ reductions that takes $[\![L]\!]_i$ to a term $N''$, which looks like $N$ with some subterms of the form $(\mathbf{rec}\ N_1)$ replaced by $(N_1\ (\cdots\ (N_1\ \Omega)))$. Hence $N' \sqsubseteq N''$. Let $M$ be the $\beta$-normal-form of $N''$, which exists because $\beta$ is strongly normalizing. By Lemma 33 and the fact that $N'$ is in normal form, we know that $N' \sqsubseteq M$. Lemma 15 now implies that $M \longrightarrow_\rho^* \ulcorner n \urcorner$.

Combining the above, we have the following situation:




where double lines indicate a partial order and vectors denotes reductions. We have thus found a normal form $M$ such that $[\![L]\!]_i$ reduces to $M$, and $M$ imperatively reduces to the final answer. This completes the proof of the auxiliary claim. ∎

A straightforward implementation of this compilation/execution schema relies on lazy evaluation. The compiler suspends after producing sufficient output and pipes its output into an abstract machine for imperative programs. When the machine runs out of executable code, it resumes the compiler. The abstract machine is a modification of the CEK machine [6]. The control portion of the machine is a member of **W**. The environment acts as a stack of references. The continuation corresponds to an evaluation context. Figure 3 contains a formal specification of the machine and its instructions.

## 4 Strong Normalization

The simply-typed $\lambda$-calculus has the important property that terms without the recursion construct always reduce to normal form. As a result, the equational theory is decidable, which is important for the implementation of a broad class of compile time optimizations. Since the imperative sub-language of IA is also clearly strongly normalizing, the natural question is whether the combined language (without **fix**) satisfies the strong normalization theorem.

The key to the Strong-Normalization Theorem for IA is (a stronger version of) the Postponement Theorem of Section 3 and a proof technique for combinations of two strongly-normalizing systems that satisfy the postponement property.[5] Appendix D contains the proof of the meta-theorem on combining

[5] Van Daalen [20, p. 80] apparently proves the same result, but he ignores the additional conditions we impose. Their absence breaks the meta-theorem.

| *Before* | | | | *After* | | |
|---|---|---|---|---|---|---|
| C | E | K | | C | E | K |
| $((op\ t_1)\ t_2)$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | $((op\ \bullet)\ t_2) :: K$ |
| $\ulcorner n \urcorner$ | $E$ | $((op\ \bullet)\ t_2) :: K$ | $\longrightarrow$ | $t_2$ | $E$ | $((op\ \ulcorner n \urcorner)\ \bullet) :: K$ |
| $\ulcorner m \urcorner$ | $E$ | $((op\ \ulcorner n \urcorner)\ \bullet) :: K$ | $\longrightarrow$ | $\ulcorner n\ op\ m \urcorner$ | $E$ | $K$ |
| | | | | | | |
| $\mathbf{new}(x, t_1).t_2$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | $\mathbf{new}(x, \bullet).t_2 :: K$ |
| $\ulcorner n \urcorner$ | $E$ | $\mathbf{new}(x, \bullet).t_2 :: K$ | $\longrightarrow$ | $t_2$ | $(x, \ulcorner n \urcorner) :: E$ | $\mathbf{new}(x, \ulcorner n \urcorner).\bullet :: K$ |
| $val$ | $p :: E$ | $\mathbf{new}(x, \ulcorner n \urcorner).\bullet :: K$ | $\longrightarrow$ | $val$ | $E$ | $K$ |
| | | | | | | |
| $t_1 := t_2$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | $\bullet := t_2 :: K$ |
| $x$ | $E$ | $\bullet := t_2 :: K$ | $\longrightarrow$ | $t_2$ | $E$ | $x := \bullet :: K$ |
| $\ulcorner n \urcorner$ | $E$ | $x := \bullet :: K$ | $\longrightarrow$ | **skip** | $E!(x, \ulcorner n \urcorner)$ | $K$ |
| | | | | | | |
| $!t_1$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | $!\bullet :: K$ |
| $x$ | $E$ | $!\bullet :: K$ | $\longrightarrow$ | $E.x$ | $E$ | $K$ |
| | | | | | | |
| **if** $t_1\ t_2\ t_3$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | **if** $\bullet\ t_2\ t_3 :: K$ |
| $\ulcorner 0 \urcorner$ | $E$ | **if** $\bullet\ t_2\ t_3 :: K$ | $\longrightarrow$ | $t_2$ | $E$ | $K$ |
| $\ulcorner n+1 \urcorner$ | $E$ | **if** $\bullet\ t_2\ t_3 :: K$ | $\longrightarrow$ | $t_3$ | $E$ | $K$ |
| | | | | | | |
| **begin** $t_1\ t_2$ | $E$ | $K$ | $\longrightarrow$ | $t_1$ | $E$ | **begin** $\bullet\ t_2 :: K$ |
| **skip** | $E$ | **begin** $\bullet\ t_2 :: K$ | $\longrightarrow$ | $t_2$ | $E$ | $K$ |

$$
\begin{aligned}
(((x, \ulcorner n \urcorner) :: E)!(x, \ulcorner m \urcorner) &= (x, \ulcorner m \urcorner) :: E \\
((y, \ulcorner n \urcorner) :: E)!(x, \ulcorner m \urcorner) &= (y, \ulcorner n \urcorner) :: (E!(x, \ulcorner m \urcorner)), \text{where } x \neq y
\end{aligned}
$$

$$
\begin{aligned}
((x, \ulcorner n \urcorner) :: E).x &= \ulcorner n \urcorner \\
((y, \ulcorner n \urcorner) :: E).x &= E.x, \text{where } x \neq y
\end{aligned}
$$

Figure 3: CEK Machine

strong-normalization results for two different systems.

**Theorem 17** *$\beta\rho$ is strongly-normalizing.*

**Proof:** Since the combination of two strongly-normalizing systems that satisfy the postponement property is not necessarily strongly-normalizing, we need to prove a technical lemma that strengthens these properties. For our case, the relevant properties are:

**finite branching** A one-step reduction relation is *finitely branching* if, for every term, the set of terms reachable in one step is finite.

**strong postponement** If $\mathbf{r}_1$ and $\mathbf{r}_2$ satisfy postponement, they satisfy *strong* postponement if $M \longrightarrow^{l}_{\mathbf{r}_1\mathbf{r}_2} M''$ implies that there exists $M'$ such that $M \longrightarrow^{m}_{\mathbf{r}_1} M' \longrightarrow^{n}_{\mathbf{r}_2} M''$, and $m + n \geq l$.[6]

Our main technical theorem (Theorem 38) is the following:

> If $\mathbf{r}_1$ and $\mathbf{r}_2$ are strongly-normalizing *and* satisfy the finite-branching property, and if $\mathbf{r}_1\mathbf{r}_2$ satisfies the strong-postponement property with respect to $\mathbf{r}_2$, then $\mathbf{r}_1\mathbf{r}_2$ is strongly normalizing.

Given this theorem, all that remains to prove that $\beta\rho$ is strongly-normalizing is to show that $\beta\rho$ satisfies the *strong* postponement property with respect to $\rho$ and that $\rho$ is strongly-normalizing. It is obvious that both $\beta$ and $\rho$ satisfy finite branching. The proof of the Postponement Theorem (7) is easily modified to show strong postponement. The following lemma shows that $\rho$ is strongly normalizing. ∎

**Lemma 18** *$\rho$ is strongly-normalizing.*

**Proof:** We note that every $\rho$ reduction removes at least one keyword, with the exception of $\sigma$, which replaces an assignment with a **skip**. Hence, any reduction starting with a term $t$ could not possibly have more that $2s + k$ steps, where $s$ is the number of assignments in $t$ and $k$ is the number of other keywords in $t$. ∎

## 5 Extensions and Alternatives

The preceding analysis of a small, but prototypical version of ALGOL formalizes a number of folklore claims. First, it proves that the language's calculus is indeed the extension of a term rewriting systems for a simple imperative language with a typed $\lambda$-calculus. Second, the combination is orthogonal in the sense that major properties for the two sub-calculi are compatible and hold for the entire system. Finally, the analysis confirms the idea that the evaluation of ALGOL programs can be neatly separated into two phases.

An extension of our results to more expressive languages than IA is possible. The analysis obviously carries over to extensions of IA that include different primitive data types (boolean, float, characters), complex data types of ground types (strings, arrays, records), and intersection types of ground types with coercions. A little extra care is needed to extend the results to a

[6] Postponement here refers to Theorem 7, not to Theorem 16.

version of IA with errors and non-local control constructs. Moreover, all the results can be re-established for a *call-by-value* variant of IA, but, for the Postponement Theorem to hold, the functional system becomes more complex and must include "bubbling" reductions for imperative operations [4, ch. 5].

The Strong-Normalization and Postponement results cannot carry over to languages with higher-typed or untyped references. As a consequence, these results do not hold for the calculi of several programming languages that mix functional and imperative features, i.e., RUSSELL [3], SCHEME (LISP) [2, 7, 8, 11, 12], and ML [22]. A recently discovered alternative to mixing functional and fully imperative languages is the addition of a weakened form of assignment to functional languages [9, 15]. None of these languages or calculi is comparable to IA with respect to (imperative) expressive power [5]. We suspect that most of these languages satisfy postponement and strong-normalization theorems, but it is not clear whether this is relevant given the weakness of their assignment statements.

## Acknowledgement

We are grateful to Robert Cartwright, Ian Mason, and John Gateley for comments on an early draft.

## References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics, revised edition. Studies in Logic and the Foundations of Mathematics*, Volume 103. North-Holland, Amsterdam, 1984.

[2] Eric Crank and Matthias Felleisen. Parameter-passing and the lambda-calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 233–245, 1991.

[3] Alan Demers and James Donahue. Making variables abstract: an equational theory for RUSSELL. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 59–72, 1983.

[4] Matthias Felleisen. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[5] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991. Preliminary version in Neil Jones, editor, *Proceedings of the European Symposium on Programming*, pages 134–151. Lecture Notes in Computer Science, 432. Springer Verlag, Berlin, 1990.

[6] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[7] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. Preliminary version in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 314–325, 1987.

[8] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:347–408, 1992. Preliminary version appeared as Technical Report 100, Rice University, Computer Science Department, 1989.

[9] Juan C. Guzman and Paul Hudak. Single-threaded polymorphic lambda-calculus. In *Proceedings of the Symposium on Logic in Computer Science*, pages 333–345, 1990.

[10] Arthur Franklin Lent. The category of functors from state shapes to bottomless CPO's is adequate for block structure. Master's thesis, Massachusetts Institute of Technology, 1992.

[11] Ian A. Mason and Carolyn Talcott. Equivalence in functional programming languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991. Preliminary version in *Proceedings of the International Conference on Automata, Languages and Programming*, pages 574–588. Lecture Notes in Computer Science, 372. Springer Verlag,

[12] Ian A. Mason and Carolyn Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992. Preliminary version in *Proceedings of the Symposium on Logic in Computer Science*, pages 284–293, 1989.

[13] Peter Naur, ed. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963. See Chapter 1.

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

[15] Uday S. Reddy, V. Swarup, and E. Ireland. Assignments for applicative languages. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 192–214. Lecture Notes in Computer Science 523. Springer Verlag, Berlin, 1991. See Chapter 9.

[16] John C. Reynolds. The essence of ALGOL. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981. See Chapter 3.

[17] John C. Reynolds. Preliminary design of the programming language FORSYTHE. Technical Report 159, Carnegie Mellon University, Computer Science Department, 1988. See Chapter 8.

[18] John C. Reynolds. Replacing complexity with generality: The programming language FORSYTHE. Carnegie Mellon University, Computer Science Department, April 1991. See Chapter 8.

[19] H. Søndergard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[20] D. van Daalen. *The Language Theory of AUTOMATH*. PhD thesis, Eindhoven University, 1980.

[21] Philip Wadler. Comprehending monads. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 61–78, 1990.

[22] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1995. Preliminary version appeared as Technical Report 160, Rice University, Computer Science Department, 1991.

## Appendix A Church-Rosser Theorem for *ia*

Let $\overrightarrow{\equiv}_\rho$ be the reflexive closure of $\longrightarrow_\rho$. We show that $\overrightarrow{\equiv}_\rho$ satisfies the diamond property, hence so does $\longrightarrow_\rho^*$.

**Lemma 19** *If $L \overrightarrow{\equiv}_\rho M$ and $L \overrightarrow{\equiv}_\rho N$, then there exists $K$ such that $M \overrightarrow{\equiv}_\rho K$ and $N \overrightarrow{\equiv}_\rho K$.*

**Proof:** If $L \equiv M$, take $K \equiv N$. Otherwise, we must have $L \equiv C[r] \longrightarrow_\rho C[c] \equiv M$, where $(r,c) \in \rho$. By symmetry, we may assume that $L \not\equiv N$. We thus have that $L \equiv C'[r'] \longrightarrow_\rho C'[c'] \equiv N$, where $(r',c') \in \rho$. If $r$ and $r'$ are in different sub-expressions of $L$ we may simply reduce $r'$ in $M$, and $r$ in $N$ to produce $K$. If $M \equiv N$ we are done, take $K \equiv M$. Because the partition of an expression into an evaluation context and a redex is unique, the only remaining case is that $r$ is a sub-expression of $r'$ or vice versa. Without loss of generality, we may assume that $r'$ is a sub-expression of $r$. We proceed by cases on $r$:

1. $r \equiv (\mathit{op}\ \ulcorner n \urcorner)\ \ulcorner m \urcorner)$, $c \equiv \ulcorner n\ \mathit{op}\ m \urcorner$. Impossible, no sub-expression of $r$ is a $\rho$ redex.
2. $r \equiv \mathbf{new}(x, \ulcorner n \urcorner).v$, $c \equiv v$. Impossible, no subexpression of $r$ is a $\rho$ redex.
3. $r \equiv \mathbf{new}(x, \ulcorner n \urcorner).E[!x]$, $c \equiv \mathbf{new}(x, \ulcorner n \urcorner).E[\ulcorner n \urcorner]$. We have that $N \equiv C[\mathbf{new}(x, \ulcorner n \urcorner).N_1]$, where $E[!x] \longrightarrow_\rho N_1$. By Lemma 20, there is an evaluation context $E'$ such that $N_1 \equiv E'[!x]$. Take $K \equiv C[\mathbf{new}(x, \ulcorner n \urcorner).E'[\ulcorner n \urcorner]]$.
4. $r \equiv \mathbf{new}(x, \ulcorner n \urcorner).E[x := \ulcorner m \urcorner]$, $c \equiv \mathbf{new}(x, \ulcorner m \urcorner).E[\mathbf{skip}]$. Similar to case 3.
5. $r \equiv \mathbf{if}\ \ulcorner 0 \urcorner\ L_1\ L_2$, $c \equiv L_1$. Either $N \equiv C[\mathbf{if}\ \ulcorner 0 \urcorner\ N_1\ L_2]$ because $L_1 \longrightarrow_\rho N_1$, or $N \equiv C[\mathbf{if}\ \ulcorner 0 \urcorner\ L_1\ N_2]$. In the first case take $K \equiv C[N_1]$, in the second $K \equiv C[L_1]$.
6. $r \equiv \mathbf{if}\ \ulcorner n+1 \urcorner\ L_1\ L_2$, $c \equiv L_2$. Similar to case 5.
7. $r \equiv \mathbf{begin\ skip}\ L_1$, $c \equiv L_1$. Similar to case 5. ∎

**Lemma 20** *If $E[z] \longrightarrow_\rho N$, where $E$ is an evaluation context and $z$ is a fresh variable, then there is an evaluation context $E'$ such that $N \equiv E'[z]$.*

**Proof:** By induction on the structure of $E$:

1. $E \equiv [\ ]$. Vacuously true.
2. $E \equiv ((\mathit{op}\ E'')\ M) \longrightarrow_\rho N$. The outermost $\delta$-redex cannot be reduced, since $E''[z]$ cannot be a numeral. Hence either $N \equiv ((\mathit{op}\ N_1)\ M)$ where $E''[z] \longrightarrow_\rho N_1$, or $N \equiv ((\mathit{op}\ E''[z])\ N_2)$ where $M \longrightarrow_\rho N_2$. In the first case we may apply the induction hypothesis to $E''$ to conclude that there exists an evaluation context $E'''$ such that $N_1 \equiv E'''[z]$. We may then take $E' \equiv ((\mathit{op}\ E''')\ M)$. In the second case we may take $E' \equiv ((\mathit{op}\ E'')\ N_2)$.

All other cases follow similarly, noting in each case that the outermost redex cannot be reduced. Thus, after applying the induction hypothesis, the desired evaluation context is easily constructed. ∎

**Definition 21** *Parallel imperative reduction* is defined as follows:

1. $M \overrightarrow{1}_\rho M$
2. If $M \longrightarrow_\rho N$, then $M \overrightarrow{1}_\rho N$
3. If $C \overrightarrow{1}_\rho C'$ and $M \overrightarrow{1}_\rho N$, then $C[M] \overrightarrow{1}_\rho C'[N]$.

For contexts, $C \overrightarrow{1}_\rho C'$ iff $C[z] \overrightarrow{1}_\rho C'[z]$, where $z$ is a fresh variable.

**Lemma 22** *If $L \longrightarrow_{\beta\mathbf{fix}}^* M$ and $L \longrightarrow_\rho^* N$, then there exists $K$ such that $M \longrightarrow_\rho^* K$ and $N \longrightarrow_{\beta\mathbf{fix}}^* K$.*

**Proof:** We show the following, and apply diagram chasing: if $L \longrightarrow_{\beta\mathbf{fix}} M$ and $L \longrightarrow_\rho N$, then there exists $K$ such that $M \xrightarrow[1]{}_\rho K$ and $N \xrightarrow{=}_{\beta\mathbf{fix}} K$. We proceed by cases on the $\beta\mathbf{fix}$ redex $r$ in $L \equiv C[r] \longrightarrow_{\beta\mathbf{fix}} C[c] \equiv M$.

1. $r \equiv ((\lambda x.L_1)\ L_2)$, $c \equiv [L_2/x]L_1$. By cases on the reduction $L \longrightarrow_\rho N$, noting that neither $L_1$ nor $L_2$ is in an evaluation context, hence any $\rho$ reduction in them must be local:
   (a) $L \longrightarrow_\rho C[((\lambda x.N_1)\ L_2)] \equiv N$ because $L_1 \longrightarrow_\rho N_1$. Take $K \equiv C[[L_2/x]N_1]$. Clearly, $N \longrightarrow_{\beta\mathbf{fix}} K$. Also, $M \xrightarrow[1]{}_\rho K$ by Lemma 23.
   (b) $L \longrightarrow_\rho C[((\lambda x.L_1)\ N_2)] \equiv N$ because $L_2 \longrightarrow_\rho N_2$. Take $K \equiv C[[N_2/x]L_1]$. Clearly, $N \longrightarrow_{\beta\mathbf{fix}} K$. Also, $M \xrightarrow[1]{}_\rho K$ by Lemma 23.
   (c) $L \longrightarrow_\rho N$ because $C[z] \longrightarrow_\rho N$. Take $K \equiv N$.
   (d) $L \longrightarrow_\rho C'[r] \equiv N$ because $C[z] \longrightarrow_\rho C'[z]$. Take $K \equiv C'[c]$.
2. $r \equiv (\mathbf{rec}\ L_1)$, $c \equiv (L_1\ (\mathbf{rec}\ L_1))$. By cases on the reduction $L \longrightarrow_\rho N$, noting that $L_1$ is not in an evaluation context, hence any $\rho$ reduction in $L_1$ must be local.
   (a) $L \longrightarrow_\rho C[(\mathbf{rec}\ N_1)] \equiv N$ because $L_1 \longrightarrow_\rho N_1$. Take $K \equiv C[(N_1\ (\mathbf{rec}\ N_1))]$.
   (b) $L \longrightarrow_\rho N$ because $C[z] \longrightarrow_\rho N$. Take $K \equiv N$.
   (c) $L \longrightarrow_\rho C'[r] \equiv N$ because $C[z] \longrightarrow_\rho C'[z]$. Take $K \equiv C'[c]$. ∎

**Lemma 23** *If* $M \xrightarrow[1]{}_\rho M'$ *and* $N \xrightarrow[1]{}_\rho N'$ *then* $[N/x]M \xrightarrow[1]{}_\rho [N'/x]M'$.

**Proof:** By induction on the proof of $M \xrightarrow[1]{}_\rho M'$ and case analysis on the last step in the proof:

1. $M \equiv M'$. By induction on the number of free $x$'s in $M$.
2. $M \equiv C[r] \longrightarrow_\rho C[c] \equiv M'$, where $(r,c) \in \rho$. All we must show is that for each imperative redex $r$, and its corresponding contractum $c$, $[N/x]r \xrightarrow[1]{}_\rho [N'/x]c$. By cases on the kind of redex $r$:
   (a) $r \equiv ((op\ \ulcorner n \urcorner)\ \ulcorner m \urcorner) \longrightarrow_\rho \ulcorner n\ op\ m \urcorner \equiv c$. Then, $[N/x]r \equiv r \xrightarrow[1]{}_\rho c \equiv [N'/x]c$.
   (b) $r \equiv \mathbf{new}(y, \ulcorner n \urcorner).v \longrightarrow_\rho v \equiv c$, where $v \equiv \ulcorner n \urcorner$ or $\mathbf{skip}$. Then, $[N/x]r \equiv r \xrightarrow[1]{}_\rho c \equiv [N'/x]c$.
   (c) $r \equiv \mathbf{new}(y, \ulcorner n \urcorner).E[!y] \longrightarrow_\rho \mathbf{new}(y, \ulcorner n \urcorner).E[\ulcorner n \urcorner] \equiv c$. By Lemma 24 and induction on the number of free $x$'s in $E$:

$$\begin{aligned} [N/x]r &\equiv \mathbf{new}(y, \ulcorner n \urcorner).([N/x]E)[!y] \\ &\xrightarrow[1]{}_\rho \mathbf{new}(y, \ulcorner n \urcorner).([N'/x]E)[\ulcorner n \urcorner] \\ &\equiv [N'/x]c \end{aligned}$$

   (d) $r \equiv \mathbf{new}(y, \ulcorner n \urcorner).E[y := \ulcorner m \urcorner]$. Similar to case 2c.
   (e) $r \equiv \mathbf{if}\ \ulcorner 0 \urcorner\ M_1\ M_2 \longrightarrow_\rho M_1 \equiv c$. By rule 3 in the definition of $\xrightarrow[1]{}_\rho$, and induction on the number of free $x$'s in $M_1$:

$$\begin{aligned} [N/x]r &\equiv \mathbf{if}\ \ulcorner 0 \urcorner\ [N/x]M_1\ [N/x]M_2 \\ &\xrightarrow[1]{}_\rho [N'/x]M_1 \\ &\equiv [N'/x]c \end{aligned}$$

   (f) $r \equiv \mathbf{if}\ \ulcorner n+1 \urcorner\ M_1\ M_2$. Similar to case 2e.
   (g) $r \equiv \mathbf{begin}\ \mathbf{skip}\ M_1$. Similar to case 2e.

3. $M \equiv C[M_1] \xrightarrow[1]{}_\rho M' \equiv C'[M_1']$, because $C \xrightarrow[1]{}_\rho C'$ and $M_1 \xrightarrow[1]{}_\rho M_1'$. By the induction hypothesis,
$$[N/x]C[z] \xrightarrow[1]{}_\rho [N'/x]C'[z]$$
and
$$[N/x]M_1 \xrightarrow[1]{}_\rho [N'/x]M_1' .$$
Hence $[N/x]M \xrightarrow[1]{}_\rho [N'/x]M'$. ∎

**Lemma 24** *For all evaluation contexts $E$, $[N/x]E$ is an evaluation context.*

**Proof:** By induction on the structure of $E$. ∎

## Appendix B Uniform Evaluation

**Lemma 25** *If $M \longrightarrow_{ia} N$ and $\pi \rhd M : \tau$, then $\pi \rhd N : \tau$.*

**Proof:** By induction on the structure of $M$ and case analysis on $M \longrightarrow_{ia} N$:

1. $M \equiv b$ or $op$ or $x$. Vacuously true, since $M$ is not a redex.
2. $M \equiv \lambda x.M_1 \longrightarrow_{ia} \lambda x.N_1 \equiv N$ because $M_1 \longrightarrow_{ia} N_1$. We know that $\pi \rhd M : \tau \to \tau'$ because $\pi[x/\tau] \rhd M_1 : \tau'$. We may apply the induction hypothesis to $M_1$ to conclude that $\pi[x/\tau] \rhd N_1 : \tau'$. Hence $\pi \rhd N : \tau \to \tau'$.
3. $M \equiv (\mathbf{rec}\ M_1)$. We distinguish two subcases:
   (a) $N = (\mathbf{rec}\ N_1), M_1 \longrightarrow_{ia} N_1$. Similar to case 2.
   (b) $N = (M_1\ (\mathbf{rec}\ M_1))$. We know that $\pi \rhd M : \tau$ because $\pi \rhd M_1 : \tau \to \tau$. Hence $\pi \rhd N : \tau$.

All other cases follow similarly, with either an application of the induction hypothesis or inspection of the types of sub-expressions in the contractum based on the types of sub-expressions in the redex. The only complicated case is that of the $\beta$-redex, which requires an additional lemma showing that substitution respects the type system. ∎

**Lemma 26** *If $\pi[x/\tau] \rhd M : \tau'$ and $\pi \rhd N : \tau$, then $\pi \rhd [N/x]M : \tau'$.*

**Proof:** By induction on the structure of $M$. ∎

**Lemma 27** *If $\{(x_1, \mathbf{int\ ref}), \ldots, (x_n, \mathbf{int\ ref})\} \rhd N : \tau$, and $N$ is a normal form, then $N \in t_1$, the language described by the following grammar:*

$$\begin{array}{lll} t_1 & ::= & \ulcorner n \urcorner \mid op \mid (op\ \ulcorner n \urcorner) \mid (op\ t_2) \mid x_i \mid \lambda y.M \mid \mathbf{skip} \mid t_2 \\ t_2 & ::= & E[!x_i] \mid E[x_i := \ulcorner n \urcorner],\ \textit{where}\ x_i \notin TR(E) \end{array}$$

**Proof:** By induction on the possible structure of the expression $N$:

1. $N \equiv \ulcorner n \urcorner$ or $op$. $N \in t_1$.
2. $N \equiv x$. Since $\{(x_1, \mathbf{int\ ref}), \ldots, (x_n, \mathbf{int\ ref})\} \rhd N : \tau$, we must have that $x \equiv x_i$, for some $i$. Hence $N \in t_1$.
3. $N \equiv \lambda y.N_1$. $N \in t_1$.
4. $N \equiv (\mathbf{rec}\ N_1)$. Impossible, since $N$ would not be in normal form.
5. $N \equiv (N_1\ N_2)$. Apply the induction hypothesis to $N_1$. Using Lemma 28 we may conclude that there are only the following possibilities, since $N_1$ is of $\to$ type.
   (a) $N_1 \equiv op$. Apply the induction hypothesis to $N_2$. Since $N_2$ is of type **int**, either $N_2 \equiv \ulcorner n \urcorner$ or $N_2 \in t_2$. In either case, $N \in n_1$.

(b) $N_1 \equiv (op\ \ulcorner n \urcorner)$. Apply the induction hypothesis to $N_2$, which is of type **int**. Either $N_2 \equiv \ulcorner m \urcorner$ or $N_2 \in t_2$. The first case is impossible, since $N$ would not be in normal form. In the second case, since $N_2$ is in an evaluation context, we have that $N \in t_2 \subset t_1$.

(c) $N_1 \equiv (op\ N_1')$, where $N_1' \in t_2$. Since $N_1'$ is in an evaluation context, we have that $N \in t_2 \subset t_1$.

(d) $N_1 \equiv \lambda y.N_1'$. Impossible, since $N$ would not be in normal form.

6. $N \equiv \mathbf{new}(x, N_1).N_2$. Apply the induction hypothesis to $N_1$, which is of type **int**.

   (a) $N_1 \equiv \ulcorner n \urcorner$. Apply the induction hypothesis to $N_2$, which is of type $\varsigma$. If $N_2 \equiv \ulcorner n \urcorner$ or **skip**, we have a contradiction with the fact that $N$ is in normal form, since **pop** could be be applied. If $N_2 \in t_2$, then either $x$ captures the free $x_i$ in $N_2$, contradicting the assumption that $N$ is in normal form, or $N \in t_2$, since $N_2$ is in an evaluation context.

   (b) $N_1 \in t_2$. Since $N_1$ is in an evaluation context, we have that $N \in t_2 \subset t_1$.

7. $N \equiv !N_1$. Apply the induction hypothesis to $N_1$, which is of type **int ref**.

   (a) $N_1 \equiv x_i$. $N \in t_2$.

   (b) $N_1 \in t_2$. Since $N_1$ is in an evaluation context, $N \in t_2$.

8. $N \equiv N_1 := N_2$. Apply the induction hypothesis to $N_1$, which is of type **int ref**.

   (a) $N_1 \equiv x_i$. Apply the induction hypothesis to $N_2$, which is of type **int**.

      i. $N_2 \equiv \ulcorner n \urcorner$. $N \in t_2$.

      ii. $N_2 \in t_2$. Since $N_2$ is in an evaluation context, $N \in t_2$.

   (b) $N_1 \in t_2$. Since $N_1$ is in an evaluation context, $N \in t_2$.

9. $N \equiv \mathbf{skip}$. $N \in t_1$.

10. $N \equiv \mathbf{if}\ N_1\ N_2\ N_3$. Apply the induction hypothesis to $N_1$, which is of type **int** and is in an evaluation context. Either $N \in t_2$, or there is a contradiction with the fact that $N$ is in normal form.

11. $N \equiv \mathbf{begin}\ N_1\ N_2$. Apply the induction hypothesis to $N_1$, which is of type **comm** and is in an evaluation context. ∎

**Lemma 28** *For all evaluation contexts $E \not\equiv [\ ]$, if $\pi \rhd E[M] : \tau$ then $\tau = o$.*

**Proof:** By simple case analysis of the structure of an evaluation context $E$. ∎

## Appendix C Postponement

**Lemma 29** *If $L \xrightarrow[1]{}_\rho M \longrightarrow_{\beta\mathbf{fix}} N$ then $\exists M'$ such that $L \longrightarrow_{\beta\mathbf{fix}} M' \xrightarrow[1]{}_\rho N$.*

**Proof:** $L \xrightarrow[1]{}_\rho M \equiv C[P] \longrightarrow_{\beta\mathbf{fix}} N \equiv C[P']$, where $(P, P') \in \beta\mathbf{fix}$. Cases on $P$:

1. $P \equiv ((\lambda x.M_1)\ M_2)$, $P' \equiv [M_2/x]M_1$. Since a $\rho$ reduction cannot create a $\beta$-redex, we must have that $L \equiv C'[((\lambda x.L_1)\ L_2)]$, where $C' \xrightarrow[1]{}_\rho C$. In addition, since neither $L_1$ nor $L_2$ is in an evaluation context, we must have that $L_i \xrightarrow[1]{}_\rho M_i$, for $i = 1, 2$. Take $M' \equiv C'[[L_2/x]L_1]$. Clearly, $L \longrightarrow_{\beta\mathbf{fix}} M'$. Also, $M' \xrightarrow[1]{}_\rho N$ by Lemma 23.

2. $P \equiv (\mathbf{rec}\ M_1)$, $P' \equiv (M_1\ (\mathbf{rec}\ M_1))$. Since a $\rho$ reduction cannot create a **fix**-redex, $L \equiv C'[(\mathbf{rec}\ L_1)]$, where $C' \xrightarrow[1]{}_\rho C$. Again, since $L_1$ is not in an evaluation context, we must have that $L_1 \xrightarrow[1]{}_\rho M_1$. Take $M' \equiv C'[(L_1\ (\mathbf{rec}\ L_1))]$. Clearly, $L \longrightarrow_{\beta\mathbf{fix}} M'$. Also, $M' \xrightarrow[1]{}_\rho N$, by rule 3 in the definition of $\xrightarrow[1]{}_\rho$. ∎

**Lemma 30** *For all programs $M$ and integers $i$, the $\beta$-normal-form of $[\![M]\!]_i$ is in* **W**.

**Proof:** Let $N$ be the $\beta$-normal-form of $[\![M]\!]_i$. We must show four things: $N$ contains no **rec**-expressions, $N$ contains no $\lambda$-expressions, all applications in $N$ are of the form $((op\ t)\ t)$, and the only place where *op* may appear is in the function position of such an application.

That there are no **rec**-expressions in $[\![M]\!]_i$ is obvious. No reduction introduces a new **rec**-expression, hence there are no **rec**-expressions in $N$.

We prove that $N$ contains no $\lambda$-expressions by contradiction. Assume not, let $L$ be the leftmost $\lambda$-expression. We consider the immediate surrounding context of $L$. It cannot be empty since a program must be of type **int**. Since $L$ is of $\rightarrow$ type it can only occur in one of three positions: body of $\lambda$-expression, function position of application, or argument position of application. The first case contradicts that $L$ is the leftmost such sub-expression. The second and third cases contradict Lemma 31.

The final condition follows from the fact that the only possible remaining expressions of $\rightarrow$ type are *op* and $(op\ t)$, and the only place where they may occur is in the function position of an application. ∎

**Lemma 31** *For any closed application expression $M$ that does not contain any* **rec**-*expressions, if $M$ is in $\beta$-normal-form, the function position of $M$ is either op or $(op\ M_1)$. Hence, the argument position is of type* **int**.

**Proof:** By induction on the structure of $M$: Write $M \equiv (M_1\ M_2)$. Since $M_1$ is of $\rightarrow$ type, it must be one of: *op*, $x$, $\lambda x.M_1'$, $(\textbf{rec}\ M_1')$, or $(M_1'\ M_1'')$. In the first case we are done. The next three cases contradict that $M$ is closed, in normal form, and contains no **rec**-expressions, respectively. In the last case, we may apply the induction hypothesis to $M_1$ to conclude that $M_1' \equiv op$, in which case we are done, or $M_1' \equiv (op\ M_1'')$, which is impossible. ∎

**Lemma 32** *For all expressions $M$, $[\![M]\!]_i \sqsubseteq [\![M]\!]_{i+1}$.*

**Proof:** By induction on the structure of $M$. ∎

**Lemma 33** *For all expressions $M, N$, not containing any* **rec**-*expressions, if $M \sqsubseteq N$ then*

$$\textit{the } \beta\textit{-normal-form of } M \sqsubseteq \textit{the } \beta\textit{-normal-form of } N$$

**Proof:** By induction on the length of the reduction sequence $M \rightarrow_\beta^* M'$, where $M'$ is the $\beta$-normal-form of $M$:

1. $M$ is in normal form. Then $N$ looks like $M$ with some $\Omega$'s replaced by other expressions. Reduce all of these expressions in $N$ to normal form, producing a term $N'$ which satisfies $M \sqsubseteq N'$. $N'$ must be in normal form, since any redex remaining would also be a redex in $M$.
2. $M \equiv C[((\lambda x.M_1)\ M_2)] \rightarrow_\beta C[[M_2/x]M_1] \equiv M'' \rightarrow_\beta^* M'$. Then it must be the case that $N \equiv C'[((\lambda x.N_1)\ N_2)]$, where $M_i \sqsubseteq N_i$ and $C \sqsubseteq C'$. We have that $N \rightarrow_\beta C'[[N_2/x]N_1] \equiv N''$. By Lemma 34, $M'' \sqsubseteq N''$, and the induction hypothesis may be applied. ∎

**Lemma 34** *If $M \sqsubseteq M'$, $N \sqsubseteq N'$, then $[N/x]M \sqsubseteq [N'/x]M'$.*

**Proof:** By induction on the structure of $M$. ∎

## Appendix D Strong Normalization

Initially, we meant to prove that the orthogonal combination of two strongly-normalizing systems is strongly-normalizing by applying van Daalen's corresponding theorem [20, p. 80]:

**Theorem 35** *If* $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *are strongly-normalizing and if* $\mathbf{r}_1\mathbf{r}_2$ *satisfies the postponement property with respect to* $\mathbf{r}_2$ *then* $\mathbf{r}_1\mathbf{r}_2$ *is strongly-normalizing.*

Unfortunately this theorem is not true. There are several problems. First, the postponement property does not imply that there is any relationship between the length of the reduction $L \longrightarrow^*_{\mathbf{r}_1\mathbf{r}_2} N$ and the length of the reduction $L \longrightarrow^*_{\mathbf{r}_1} M \longrightarrow^*_{\mathbf{r}_2} N$. In order to solve this problem, we introduce a stronger notion of postponement, which places a lower bound on the length of the generated reduction.

**Definition 36** $\mathbf{r}_1\mathbf{r}_2$ satisfies the *strong-postponement property with respect to* $\mathbf{r}_2$, if $e \longrightarrow^l_{\mathbf{r}_1\mathbf{r}_2} M''$ implies there exists $M'$ such that $e \longrightarrow^m_{\mathbf{r}_1} M' \longrightarrow^n_{\mathbf{r}_2} M''$, and $m + n \geq l$.

Problems can also arise when $\mathbf{r}_1$ or $\mathbf{r}_2$ allows arbitrary length (not infinite) reductions for a given term. At first glance, arbitrarily long reductions starting from a single term might appear to directly contradict strong-normalization; however, in general, this is not the case. As an example, consider the following notion of reduction over the language $\Sigma^*$, with alphabet $\Sigma = \{0, 1, 2\}$,

$$\mathbf{r} = \{(0, 1^n) \mid n \geq 1\} \cup \{(1, 2)\}$$

Notice that $\mathbf{r}$ is strongly-normalizing, but for any term containing a 0, there are reductions of arbitrary length.

Intuitively, for more "standard" reduction systems such as $\beta$, $\rho$, and *ia*, strong normalization coincides with bounded reduction length. In comparing these systems with $\mathbf{r}$, we notice that they share a property that $\mathbf{r}$ does not have: only a finite number of reductions are applicable to any given term. The notion $\mathbf{r}$ does *not* satisfy this finite-branching property, because for a term $t$ containing a 0, there are an infinite number of terms $t'$ such that $t \longrightarrow_{\mathbf{r}} t'$. After introducing some terminology, we will provide a simple criterion for strongly-normalizing systems that prohibit arbitrary-length reductions starting with a given term.

**Definition 37** Let $\mathbf{r}$ be a notion of reduction.

- The set of terms *reachable* in $i$ steps from expression $M$ using notion of reduction $\mathbf{r}$ is defined by:
$$R^i_{\mathbf{r}}(M) \stackrel{df}{\equiv} \{M' \mid M \longrightarrow_{\mathbf{r}}{}^i M'\}$$
- The set of terms reachable from $M$ is defined by:
$$R_{\mathbf{r}}(M) \stackrel{df}{\equiv} \bigcup_{i \geq 0} R^i_{\mathbf{r}}(M)$$
- $\mathbf{r}$ satisfies the *finite reachability* property if for all expressions $M$, $R_{\mathbf{r}}(M)$ is finite.
- $\mathbf{r}$ satisfies the *finite branching* property if for all expressions $M$, $R^1_{\mathbf{r}}(M)$ is finite.
- $\mathbf{r}$ satisfies the *bounded reduction length* property if for all expressions $M$, there exists a $j$, such that $\bigcup_{i>j} R^i_{\mathbf{r}}(M) = \emptyset$. For an expression $M$, we denote the smallest such $j$ by $\mu_{\mathbf{r}}(M)$.

With some technical lemmas, we can show that in the presence of finite branching, strong-normalization exactly corresponds to bounded reduction length. Then we can prove the following theorem.

**Theorem 38** *If* $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *are strongly-normalizing and satisfy the finite branching property, and if* $\mathbf{r}_1\mathbf{r}_2$ *satisfies the strong-postponement property with respect to* $\mathbf{r}_2$*, then* $\mathbf{r}_1\mathbf{r}_2$ *is strongly normalizing.*

First, we need to state two obvious properties of our notion of reachability:

D.1 $R_{\mathbf{r}}(M) = M \cup \left(\bigcup_{M' \in R^1_{\mathbf{r}}(M)} R_{\mathbf{r}}(M')\right)$.

D.2 Bounded reduction length implies strong normalization.

Based on these, we now can prove the necessary lemmas for Theorem 38.

**Proposition 39** *Strong normalization plus finite branching implies finite reachability.*

**Proof:** We use infinite reachability and finite branching to generate an infinite reduction sequence. Let $M$ be a term that violates finite reachability. By Observation D.1 and finite branching we may conclude that there exists an $M' \in R^1_{\mathbf{r}}(M)$ that violates finite reachability. Apply a similar argument to $M'$. Continue. ∎

**Proposition 40** *Strong normalization plus finite reachability implies bounded reduction length.*

**Proof:** In fact, $\mu_{\mathbf{r}}(M) \leq |R_{\mathbf{r}}(M)|$. Any longer reduction would necessarily repeat a term, contradicting strong-normalization. ∎

And finally we are ready to prove our main theorem.

**Proof:** (Theorem 38) We show that $\mathbf{r}_1\mathbf{r}_2$ satisfies bounded reduction length, thus by Observation D.2 is strongly-normalizing. By Propositions 39 and 40 we know that $\mathbf{r}_1$ and $\mathbf{r}_2$ satisfy bounded reduction length. We show that $\mu_{\mathbf{r}_1\mathbf{r}_2}(M) \leq m' + n'$, where $m' = \mu_{\mathbf{r}_1}(M)$, $n' = max\{\mu_{\mathbf{r}_2}(M') \mid M' \in R_{\mathbf{r}_1}(M)\}$. We note that $n'$ is well-defined because $\mathbf{r}_1$ satisfies finite reachability (by Proposition 39) and $\mathbf{r}_2$ satisfies bounded reduction length. Consider a reduction $M \longrightarrow^{l}_{\mathbf{r}_1\mathbf{r}_2} M''$. Because $\mathbf{r}_1\mathbf{r}_2$ satisfies the strong postponement property with respect to $\mathbf{r}_2$, there exists a reduction $M \longrightarrow^{m}_{\mathbf{r}_1} M' \longrightarrow^{n}_{\mathbf{r}_2} M''$ with $m + n \geq l$. We know that $m \leq m'$. Since $M' \in R_{\mathbf{r}_1}(M)$ we also know that $n \leq \mu_{\mathbf{r}_2}(M') \leq n'$. Hence $l \leq m + n \leq m' + n'$. ∎