On the Orthogonality of Assignments and Procedures in Algol

Stephen Weeks* Matthias Felleisen†

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

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 λ-calculus. In an attempt to formalize the claim of "orthogonality", we define a simple version of Algol and an extended λ-calculus. The calculus includes the *full* β-rule and rules for the reduction of assignment statements and commands. It has the usual properties, e.g., it satisfies a Church-Rosser and Strong Normalization Theorem. In support of the claim that the imperative and functional components are orthogonal to each other, we show that the proofs of these theorems are combinations of separate Church-Rosser and Strong Normalization theorems for each sublanguage.

An acclaimed consequence of Algol's orthogonal design is the idea that the evaluation of a program has two distinct phases. The first phase corresponds to an unrolling of the program according to the usual β and fixpoint reductions, which provide the formal counterpart to Algol's famous copy rule. The result of this phase is essentially an imperative program. The second phase executes the output of the first phase in the imperative fashion of a stack machine. Given our calculus, we can prove a Postponement Theorem and can thus formalize this phase separation.

1 The Origins of Algol

According to folklore, Algol 60 "orthogonally" extends a simple imperative programming language with a typed λ-calculus. The underlying imperative language usually consists of assignment statements, branching statements, loops, and statement sequences for computing basic arithmetic algorithms. 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 successor to 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. [17:3]

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. [17:3]

Given the recent interest in integrating variations of assignment statements into functional programming languages in a controlled manner and

*New address: Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3891

†Both authors were supported in part by NSF grants CCR 89-17022 and CCR 91-22518.

To appear in:
20th POPL
January 10–13, 1993
Charleston, South Carolina

in the design of simple logics for such mixed languages [2, 9, 10, 12, 13, 16, 21], these folklore claims about Algol's design clearly deserve a rigorous analysis. On one hand, such an analysis will enhance our understanding of the role of "orthogonality" in Algol in the same manner in which Søndergard's and Sestoft's paper [20] clarified the often misused terminology of "referential transparency", and in which Felleisen's work [6] defined the idea of "expressiveness of programming languages." On the other hand, the results are a contribution to the construction of simple logics for mixed functional-imperative programming languages and to their abstract implementation.

The first step for such an analysis clearly requires the formulation of a small Algol-like language and its λ-calculus. The calculus we develop includes the *full* β-rule, despite the presence of assignments, and has the usual properties, i.e., Church-Rosser, Standardization, and Strong Normalization for the recursion-free subset. In support of the claim that the two sub-languages are orthogonal to each other, we show that the proofs of Church-Rosser and Strong Normalization, the most interesting properties of the calculus, consist of two separate subproofs, one for each sub-language, and a proof that the two systems mix smoothly. Most importantly, we confirm Reynolds's second conjecture with a Postponement Theorem, which shows how the evaluation of a program can indeed be separated into a functional phase followed by an imperative phase.

We begin by defining the syntax, calculus, and semantics of a simple Algol-like language. Section 2.3 addresses the basic consistency results for the calculus and the derived semantics. Section 3 contains the Postponement Theorem. In Section 4, we prove that Postponement combined with Strong Normalization for each sub-calculus entails Strong Normalization for the complete calculus. We address the shortcomings of our dialect in the last section; we also include a brief discussion of related work.

Note: Except for the strong-normalization theorem, the paper does not include full proofs. We refer the interested reader to our technical report 92-193 (anonymous ftp: titan.cs.rice.edu in public/languages) which contains the full proofs of the remaining theorems.

2 Idealized Algol

Our version of Idealized Algol (IA) is a simplification of Forsythe, a generalized version of Algol 60 [17, 18, 19]. Specifically, the imperative sublanguage of IA only contains an arithmetic expression language for numerals and a small set of total primitive functions. The language is simply typed, and, for simplicity, excludes intersection types and coercions, complex data, and non-local control operators. Thus, it is simple enough to permit an easily comprehensible semantics, yet complete enough to permit a generalization of our results to other Algol-like languages.

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 λ-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 evaluator.

2.1 Syntax and Informal Semantics

The definition of the syntactic part of the language proceeds in two stages: the general syntax of commands and expressions, and a type system for filtering out *legal* commands and expressions.

Syntax. Figure 1 specifies the set of syntactically feasible commands and expressions. The first part of the set of terms (above the line) constitutes the raw syntax of the imperative sub-language; the second part (below the line) is the λ-calculus extension, including a rec-construct for declaring recursive objects (procedures and the "diverging integer"). Both $\lambda x.M$ and $\mathsf{new}(x,N).M$ *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 subterm; $C[\ ]$ 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 Baren-

Syntax:

$$
\begin{array}{lll}
M \;::=\; & \ulcorner n \urcorner \mid op \mid x \mid (M\ M) & \text{arithmetic expressions} \\
 & & \text{where } n \in \mathbb{Z} \quad . \\
 & & \phantom{\text{where }} op \in \{+, -, *\} \\
 & & \phantom{\text{where }} x \in \mathit{Vars} \\
 & \mid \mathsf{skip} \mid (\mathsf{if0}\ M\ M\ M) \mid (\mathsf{begin}\ M\ M) & \text{nop, branching, sequencing} \\
 & \mid \mathsf{new}(x, M).M & \text{block} \\
 & \mid (\mathsf{deref}\ M) \mid (\mathsf{setref!}\ M\ M) & \text{dereferencing, assignment} \\
\hline
 & \mid \lambda x.M \mid (\mathsf{rec}\ M) & \text{(recursive) procedures}
\end{array}
$$

Type System:

- Constants

$$\frac{}{\pi \ \triangleright\ \ulcorner n \urcorner : \mathsf{int}}$$

$$\frac{}{\pi \ \triangleright\ op : \mathsf{int} \to \mathsf{int} \to \mathsf{int}}$$

- Variables

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

- Functions

$$\frac{\pi[x/\tau] \ \triangleright\ M : \tau'}{\pi \ \triangleright\ \lambda x.M : \tau \to \tau'}$$

$$\frac{\pi \ \triangleright\ M : \tau \to \tau}{\pi \ \triangleright\ (\mathsf{rec}\ M) : \tau}$$

$$\frac{\pi \ \triangleright\ M : \tau' \to \tau,\ \ \pi \ \triangleright\ N : \tau'}{\pi \ \triangleright\ (M\ N) : \tau}$$

- References

$$\frac{\pi \ \triangleright\ M : \mathsf{int},\ \ \pi[x/\mathsf{int\ ref}] \ \triangleright\ N : \varsigma}{\pi \ \triangleright\ \mathsf{new}(x, M).N : \varsigma}$$

$$\frac{\pi \ \triangleright\ M : \mathsf{int\ ref}}{\pi \ \triangleright\ (\mathsf{deref}\ M) : \mathsf{int}}$$

$$\frac{\pi \ \triangleright\ M : \mathsf{int\ ref},\ \ \pi \ \triangleright\ N : \mathsf{int}}{\pi \ \triangleright\ (\mathsf{setref!}\ M\ N) : \mathsf{comm}}$$

- Simple Commands & Expressions

$$\frac{}{\pi \ \triangleright\ \mathsf{skip} : \mathsf{comm}}$$

$$\frac{\pi \ \triangleright\ L : \mathsf{int},\ \ \pi \ \triangleright\ M : o,\ \ \pi \ \triangleright\ N : o}{\pi \ \triangleright\ (\mathsf{if0}\ L\ N\ N) : o}$$

$$\frac{\pi \ \triangleright\ M : \mathsf{comm},\ \ \pi \ \triangleright\ N : o}{\pi \ \triangleright\ (\mathsf{begin}\ M\ N) : o}$$

FIGURE 1: Syntax and Type Inference System for IA

dregt's [1] λ-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. ∎

Types, Legal Syntax. The type system filters legal from illegal terms and separates the set of terms into a sub-language of commands (of comm) and expressions. It is an adaptation of the type system for Forsythe [19].

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 whose (maximal) sub-expressions are of ground type, e.g.,

$$(\textsf{begin skip } ((+ \ulcorner 1 \urcorner)\ \ulcorner 3 \urcorner))$$

is a command in the imperative sub-language while

$$(\textsf{begin skip } (+ \ulcorner 1 \urcorner))$$

is not. Like Algol and Forsythe, IA restricts references to contain only basic data values (int's). Thus,

$$\textsf{new}(x, \ulcorner 0 \urcorner).(\textsf{setref!}\ x\ ((+\ (\textsf{deref}\ x))\ (\textsf{deref}\ x)))$$

is a block of type command.

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 $\tau \rightarrow \tau$ for arbitrary types τ, e.g., $\textsf{int} \rightarrow \textsf{int} \rightarrow \textsf{comm}$ is a permissible type now. The system does not lift the restriction that the result of new-blocks are of ground type.

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

$$\begin{array}{lcl} \tau & ::= & o \mid \tau \rightarrow \tau \\ o & ::= & \textsf{comm} \mid \textsf{int} \mid \textsf{int ref} \\ \varsigma & ::= & \textsf{comm} \mid \textsf{int} \end{array}$$

The type inference system is defined by a set of *inference rules*, whose basic components are *typings*. A typing, $\pi \ \triangleright\ M : \tau$ consists of a *type assignment*, π, an expression, M, and a type, τ. A type assignment, π, 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 $\pi \ \triangleright\ M : \tau$ asserts that expression M has type τ when the free variables in M are assigned types by π. 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 if0-expressions and begin-expressions are commands as well as expressions, depending on which ground type they are assigned. In addition, the requirement that if0-expressions and begin-expressions be of type o is only to simplify the theory, and may be removed in practice with the following abbreviations:

$$\begin{array}{lcl} (\textsf{if0}\ L\ M\ N) & \stackrel{df}{\equiv} & \lambda\vec{x}.(\textsf{if0}\ L\ (M\ \vec{x})\ (N\ \vec{x})) \\ (\textsf{begin}\ M\ N) & \stackrel{df}{\equiv} & \lambda\vec{x}.(\textsf{begin}\ M\ (N\ \vec{x})); \end{array}$$

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 $(N\ \vec{x})$.

Informal Semantics, Pragmatics. The behavior 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 [14]. 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:

$$\begin{array}{l} (\textsf{while}\ M\ N) \stackrel{df}{\equiv} \\ \quad (\textsf{rec}\ \lambda w.\lambda x.\lambda y. \\ \qquad\qquad (\textsf{if0}\ x\ \textsf{skip}\ (\textsf{begin}\ y\ ((w\ x)\ y)))) \end{array}$$

Other loops are similar abbreviations.

The following phrase is a simple procedure, mixing functional and imperative facilities:

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

The procedure passes the type inference system as an expression of type $\textsf{int} \rightarrow \textsf{int}$, that is, as a procedure from integers to integers. It implements a simple imperative algorithm for computing the sum from zero to its parameter i. We refer to the above procedure as $\Sigma_!$.

An equivalent functional implementation of $\Sigma_!$ is the following (pure) IA phrase:

$$(\mathsf{rec}\ \lambda sum.\lambda i.(\mathsf{if0}\ i\ \ulcorner 0 \urcorner\ ((+\ i)\ (sum\ ((-\ i)\ \ulcorner 1 \urcorner)))))$$

We use Σ_λ in the following sections to refer to this procedure.

2.2 Semantics and Calculus

To formalize IA's semantics, we define an equational calculus.[1] 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 λ_n-R-calculus for a call-by-name language and that of the λ_v-B-calculus for a language with first class reference cells [2, 9].

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 $\longrightarrow^*_{\mathbf{r}}$ is the reflexive transitive closure of $\longrightarrow_{\mathbf{r}}$. The relation $=_{\mathbf{r}}$ is the equivalence relation generated by $\longrightarrow_{\mathbf{r}}$. 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 their associated bound references. The following grammar simultaneously defines evaluation contexts and the corresponding set of trapped references, TR(E):

E ::=	TR(E)
$\mid [\,]$	$\emptyset$
$\mid ((op\ E)\ M)$	TR(E)
$\mid ((op\ \ulcorner n \urcorner)\ E)$	TR(E)
$\mid \mathsf{new}(x,E).M$	TR(E)
$\mid \mathsf{new}(x,\ulcorner n \urcorner).E$	$\{x\} \cup$ TR(E)
$\mid (\mathsf{deref}\ E)$	TR(E)
$\mid (\mathsf{setref!}\ E\ M)$	TR(E)
$\mid (\mathsf{setref!}\ x\ E)$	TR(E)
$\mid (\mathsf{if0}\ E\ M\ M)$	TR(E)
$\mid (\mathsf{begin}\ E\ M)$	TR(E)

The hole in an evaluation context indicates the sub-expression whose assignment commands and dereferencing expression 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 behavior 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 λ-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* β-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. Thus, assignments to distinct references may proceed in parallel. For an example, consider the following phrase:

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

[1] In contrast, Lent [11] defines an operational semantics of an IA-like language based on structural operational semantics. His major concern is an "adequacy theorem" for his language with respect to a semantic model.

(δ)	$((op\ \ulcorner n \urcorner)\ \ulcorner m \urcorner)$	$\longrightarrow$	$\ulcorner n\ op\ m \urcorner$	
$(\mathbf{pop})$	$\mathsf{new}(x, \ulcorner n \urcorner).v$	$\longrightarrow$	$v,$	$v \equiv \ulcorner n \urcorner$ or skip
(D)	$\mathsf{new}(x, \ulcorner n \urcorner).E[(\mathsf{deref}\ x)]$	$\longrightarrow$	$\mathsf{new}(x, \ulcorner n \urcorner).E[\ulcorner n \urcorner],$	$x \notin \mathrm{TR}(E)$
(σ)	$\mathsf{new}(x, \ulcorner n \urcorner).E[(\mathsf{setref!}\ x\ \ulcorner m \urcorner)]$	$\longrightarrow$	$\mathsf{new}(x, \ulcorner m \urcorner).E[\mathsf{skip}],$	$x \notin \mathrm{TR}(E)$
$(\mathbf{T})$	$(\mathsf{if0}\ \ulcorner 0 \urcorner\ M\ N)$	$\longrightarrow$	M	
$(\mathbf{F})$	$(\mathsf{if0}\ \ulcorner n+1 \urcorner\ M\ N)$	$\longrightarrow$	N	
$(\mathbf{B})$	$(\mathsf{begin}\ \mathsf{skip}\ M)$	$\longrightarrow$	M	
(β)	$((\lambda x.M)\ N)$	$\longrightarrow$	$M[x{:=}N]$	
$(\mathbf{fix})$	$(\mathsf{rec}\ M)$	$\longrightarrow$	$(M\ (\mathsf{rec}\ M))$	

FIGURE 2: Basic Notions of Reduction

Here all side-effects in the two distinct procedure calls of $\Sigma_!$ can proceed independently. The current calculus does not support the permuted reduction of all independent assignments or dereference expressions.

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, β**fix**, corresponds to the axioms describing the functional part of the language. The third, ρ, corresponds to the axioms describing the command component of the language:

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

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*, β**fix**, or ρ.

Given the calculus, we can define the behavior of a program in a precise manner. A program, which must be a closed integer expression, produces a result if and only if it is provably equal to a numeral. The latter is the result of the program.

Definition 2.1. (*Programs, Evaluation*) A *program* M is a closed expression of type int, i.e.,

$$M \in \mathit{Programs} \text{ if and only if } \emptyset \ \triangleright\ M : \mathsf{int}.$$

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

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

If M is a program, then $\mathit{eval}(M) = \ulcorner n \urcorner$ if and only if $ia \vdash M = \ulcorner n \urcorner$. 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 sub-languages and carry over to IA in a uniform manner.

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

Theorem 2.2 *Let M be a program, let $n, m \in \mathbb{Z}$.*

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

Proof. Both parts are immediate consequences of Theorem 2.3. ∎

Theorem 2.3 (Church-Rosser) *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. ρ and β**fix**, separately. The third lemma shows that the subsystems merge smoothly.

1. The notion of reduction $\beta\mathbf{fix}$ is Church-Rosser. This fact immediately follows from the Church-Rosser Theorem for simply-typed λ-calculus (with rec). The new syntax does not interfere with the proof.

2. The reflexive-compatible closure of ρ satisfies the diamond property directly. Hence, by a diagram chase, the reflexive, transitive closure of $\longrightarrow_\rho$ satisfies the diamond property, and ρ is Church-Rosser.

3. The two reductions generated by $\beta\mathbf{fix}$ and ρ commute. This is shown by defining a parallel reduction relation $\underset{1}{\twoheadrightarrow}_\rho$, and showing that $\longrightarrow_{\beta\mathbf{fix}}$ and $\underset{1}{\twoheadrightarrow}_\rho$ commute.

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

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.[2]

Theorem 2.4 (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 2.5) shows that reductions preserve types, and hence that a program always reduces to a program. The proof of the second (Lemma 2.6) provides a characterization of certain normal forms, and in particular shows that a program in normal form must be a numeral. ∎

The first of the two auxiliary lemmas states that reductions preserve the type of programs and more generally of arbitrary expressions.

Lemma 2.5 *If $M \longrightarrow_{ia} N$ and $\pi \ \triangleright\ M : \tau$, then $\pi \ \triangleright\ N : \tau$.*

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. ∎

The second lemma shows that a program in normal form is a numeral.

Lemma 2.6 *If P is a program in normal form, i.e., 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: if

$$\{(x_1, \mathsf{int\ ref}), \ldots, (x_n, \mathsf{int\ ref})\} \ \triangleright\ N : \tau,$$

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

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

The specific result follow by noting that a program M must satisfy $\emptyset \ \triangleright\ M : \mathsf{int}$. ∎

Theorems 2.2 and 2.4, together with Theorem 2.3, 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

As discussed in the introduction, one of Algol's design goals is a phase separation of the evaluation of programs. For IA, the first phase eliminates procedures and their uses by reducing programs with $\beta\mathbf{fix}$ 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 ρ-rules or on a regular stack machine. The following theorem makes this idea precise.

[2] 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.

Theorem 3.1 *Let M be a program, let $n \in \mathbb{Z}$. Then, $eval(M) = \ulcorner n \urcorner$ iff for some program N,*

$$M \longrightarrow^*_{\beta\mathbf{fix}} N \longrightarrow^*_{\rho} \ulcorner n \urcorner$$

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, $\underset{1}{\longrightarrow\!\!\!\!\!\rightarrow}_{\rho}$, which contains $\longrightarrow_{\rho}$ but is contained in $\longrightarrow^*_{\rho}$. This follows the method of Tait/Løf for the Church-Rosser Theorem. The second step is to replace all $\longrightarrow_{\rho}$ steps with $\underset{1}{\longrightarrow\!\!\!\!\!\rightarrow}_{\rho}$ steps in the reduction $M \longrightarrow^*_{ia} \ulcorner n \urcorner$, which must exist since $eval(M) = \ulcorner n \urcorner$. The final step is to prove that we can slide all $\longrightarrow_{\beta\mathbf{fix}}$ to the left and all $\underset{1}{\longrightarrow\!\!\!\!\!\rightarrow}_{\rho}$ to the right. The proof method is inspired by Plotkin's proof of standardization for the untyped λ-calculus [15:140]. ∎

Conceptually, this theorem says that we may view compilation as reduction in the functional fragment and execution as reduction in the imperative fragment. However, as Reynolds also points out, recursion presents a major obstacle since

> 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. [17:3]

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 extend the language of IA with an additional constant, Ω, denoting observable nontermination. We define the function $[\![\cdot]\!]_i$, which maps an expression in IA to an expression in extended IA by "unrolling" all recursive functions i times. We then define a purely imperative sublanguage of extended IA that precisely characterizes the β normal forms of unrolled programs. This imperative sublanguage may be viewed as the target language of the compiler.

Extended IA is naturally ordered by, $\sqsubseteq$, the prefix-ordering with respect to Ω. Unrolling and β-normalization respect this ordering, *i.e.*, unrolling an expression further and β normalizing the result yields a (potentially) larger member of the imperative sublanguage. Similarly, execution in the imperative sublanguage respects the ordering. Finally, we prove a version of the postponement theorem for unrolled programs.

Definition 3.2. (*Unrolling*) The ith *unrolling* of an IA expression M, notation: $[\![M]\!]_i$, is defined by induction on the structure of M. The only interesting clause is the following:

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

For all other constructs, the *unrolling* expansion is a homomorphic map. We refer to i as the unrolling index. The constant Ω 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,

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

but

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

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

Definition 3.3. ($\sqsubseteq$)

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 $C[\Omega] \sqsubseteq C'[\Omega]$. ∎

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 3.4. **(W)** The imperative sub-language **W** of extended IA is defined by the following grammar:

$$
\begin{array}{rcl}
t & ::= & \Omega \mid \ulcorner n \urcorner \mid x \mid ((op\ t)\ t) \\
 & & \mid \mathsf{new}(x,t).t \mid (\mathsf{deref}\ t) \mid (\mathsf{setref!}\ t\ t) \\
 & & \mid \mathsf{skip} \mid (\mathsf{if0}\ t\ t\ t) \mid (\mathsf{begin}\ t\ t)
\end{array}
$$

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

In order to show that **W** is the proper target language, we prove that the result of β-reducing an unrolled program to β normal form is always a member of **W**.

Lemma 3.5 *For all $i \in \mathbb{N}$, $L \in IA$, the β normal form of $[\![L]\!]_i$ is in* **W**.

Proof. The proof reduces to showing that both the function and argument position of an application in a normal form expression cannot be a λ-expression. ∎

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 β 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 2.1) to *sets of* **W** *programs.*

Definition 3.6. (*Extended $eval_\rho$*) 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 point-wise extension of the original evaluator:

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

(For completeness, we define that $\Omega \longrightarrow_\rho \Omega$.) ∎

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 β 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 β normal form of an IA program.[3] 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 3.7 *For all programs M and $i \in \mathbb{N}$, if L_j is the β normal form of $[\![M]\!]_j$, then for all $i \in \mathbb{N}$,*

$$L_i \sqsubseteq L_{i+1}$$

Proof. The result follows from two lemmas. The first shows that $[\![M]\!]_i \sqsubseteq [\![M]\!]_{i+1}$. The second shows that β normalization respects $\sqsubseteq$. ∎

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

Lemma 3.8 *For all $M, N \in$ **W**, if $M \sqsubseteq N$ and $M \longrightarrow_\rho^* \ulcorner n \urcorner$, then $N \longrightarrow_\rho^* \ulcorner n \urcorner$.*

Proof. By induction on the length of the reduction $M \longrightarrow_\rho^* \ulcorner n \urcorner$: For a single step $M \longrightarrow_\rho M_1$, since $M \sqsubseteq N$, either $M \equiv M_1$ if it is an Ω 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 β-normalization to an infinite tree and machine execution as an evaluation of the infinite tree in the imperative fragment.

Theorem 3.9 *For all IA programs L, let*

$$W = \{M \mid M \text{ is } \beta \text{ normal form of } [\![L]\!]_i \text{ for } i \in \mathbb{N}\}.$$

Then,

$$\{eval(L)\} = eval_\rho(W).$$

Proof. By Lemma 3.7, W is a chain, i.e., a totally ordered set. Thus, the theorem reduces to the following claim:

$$eval(L) = \ulcorner n \urcorner$$

[3] This situation is analogous to domain theory [1]: infinite terms are really sets of all their finite approximations.

if and only if there exists some unrolling index i and β normal form M such that

$$[\![L]\!]_i \longrightarrow^*_\beta M \longrightarrow^*_\rho \ulcorner n \urcorner$$

If this claim is true, then, by Lemma 3.8, 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 3.1, $eval(L) = \ulcorner n \urcorner$ implies that there is a reduction:

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

Clearly, no rec-expression or λ-expression in N is relevant to the reduction from N to $\ulcorner n \urcorner$. Hence we may replace all rec-expressions and λ-expressions in N by Ω to produce a term N' such that $N' \sqsubseteq N$ and $N' \longrightarrow^*_\rho \ulcorner n \urcorner$. Next, let i be the number of $\mathbf{fix}$ steps in the reduction from L to N. Directly corresponding to the sequence of β reductions in $L \longrightarrow^*_{\beta\mathbf{fix}} N$ is a sequence of β reductions that takes $[\![L]\!]_i$ to a term N'', which looks like N with some subterms of the form $(\mathsf{rec}\ N_1)$ replaced by $(N_1\ (\cdots\ (N_1\ \Omega)))$. Hence $N' \sqsubseteq N''$. Let M be the β normal form of N'', which exists because β is strongly normalizing. Because β normalization respects $\sqsubseteq$, we know that $N' \sqsubseteq M$. Hence, by Lemma 3.8, we have $M \longrightarrow^*_\rho \ulcorner n \urcorner$.

Combining the above, we have the following situation:

$$\begin{tikzcd}
L \arrow[r, "\beta\mathbf{fix}"'] & N \arrow[r, "\rho"'] \arrow[d, equal] & \ulcorner n \urcorner \\
& N' \arrow[ur, "\rho"] \arrow[dr, equal] \arrow[d, equal] & \\
[\![L]\!]_i \arrow[r, "\beta"'] & N'' \arrow[r, "\beta"'] & M \arrow[uu, "\rho"']
\end{tikzcd}$$

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 [7]. The control portion of the machine is a member of $\mathbf{W}$. 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 λ-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 clearly 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 $\mathbf{fix}$) 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.[4] Appendix A contains the proof of the meta-theorem on combining strong normalization results for two different systems.

Theorem 4.1 *$\beta\rho$ 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 there exists M' such that $M \longrightarrow^m_{\mathbf{r}_1} M' \longrightarrow^n_{\mathbf{r}_2} M''$, and $m + n \geq l$.[5]

[4] Van Daalen [3:80] apparently proves the same result, but he ignores the additional conditions we impose. Their absence breaks the meta-theorem.

[5] Postponement refers to Theorem 3.1 not to Theorem 3.9.

Before				*After*		
C	E	K		C	E	K
$((op\ t_1)\ t_2)$	E	K	$\longrightarrow$	t_1	E	$((op\ *)\ t_2) :: K$
$\ulcorner n \urcorner$	E	$((op\ *)\ t_2) :: K$	$\longrightarrow$	t_2	E	$((op\ \ulcorner n \urcorner)\ *) :: K$
$\ulcorner m \urcorner$	E	$((op\ \ulcorner n \urcorner)\ *) :: K$	$\longrightarrow$	$\ulcorner n\ op\ m \urcorner$	E	K
$\mathsf{new}(x,t_1).t_2$	E	K	$\longrightarrow$	t_1	E	$\mathsf{new}(x,*).t_2 :: K$
$\ulcorner n \urcorner$	E	$\mathsf{new}(x,*).t_2 :: K$	$\longrightarrow$	t_2	$(x, \ulcorner n \urcorner) :: E$	$\mathsf{new}(x,\ulcorner n \urcorner).* :: K$
val	$p :: E$	$\mathsf{new}(x,\ulcorner n \urcorner).* :: K$	$\longrightarrow$	val	E	K
$(\mathsf{setref!}\ t_1\ t_2)$	E	K	$\longrightarrow$	t_1	E	$(\mathsf{setref!}\ *\ t_2) :: K$
x	E	$(\mathsf{setref!}\ *\ t_2) :: K$	$\longrightarrow$	t_2	E	$(\mathsf{setref!}\ x\ *) :: K$
$\ulcorner n \urcorner$	E	$(\mathsf{setref!}\ x\ *) :: K$	$\longrightarrow$	skip	$E!(x, \ulcorner n \urcorner)$	K
$(\mathsf{deref}\ t_1)$	E	K	$\longrightarrow$	t_1	E	$(\mathsf{deref}\ *) :: K$
x	E	$(\mathsf{deref}\ *) :: K$	$\longrightarrow$	$E.x$	E	K
$(\mathsf{if0}\ t_1\ t_2\ t_3)$	E	K	$\longrightarrow$	t_1	E	$(\mathsf{if0}\ *\ t_2\ t_3) :: K$
$\ulcorner 0 \urcorner$	E	$(\mathsf{if0}\ *\ t_2\ t_3) :: K$	$\longrightarrow$	t_2	E	K
$\ulcorner n+1 \urcorner$	E	$(\mathsf{if0}\ *\ t_2\ t_3) :: K$	$\longrightarrow$	t_3	E	K
$(\mathsf{begin}\ t_1\ t_2)$	E	K	$\longrightarrow$	t_1	E	$(\mathsf{begin}\ *\ t_2) :: K$
skip	E	$(\mathsf{begin}\ *\ t_2) :: K$	$\longrightarrow$	t_2	E	K

where the operations ! and . on environments are defined as follows:

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

FIGURE 3: CEK Machine

Our main technical theorem (Theorem A.4) 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, to prove that $\beta\rho$ is strongly-normalizing, all that remains to be shown is that $\beta\rho$ satisfies the *strong* postponement property with respect to ρ and that ρ is strongly-normalizing. It is obvious that both β and ρ satisfy finite branching. The proof of the postponement theorem (3.1) is easily modified to show strong postponement. The following lemma (Lemma 4.2) shows that ρ is strongly normalizing. ∎

Lemma 4.2 *ρ is strongly-normalizing.*

Proof. We note that every ρ reduction removes at least one keyword, with the exception of σ, which replaces a **setref!** 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 **setref!**'s 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 λ-calculus. Second, the combination is orthogonal in the sense that major properties for the two subcalculi 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 a functional and an imperative phase.

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. 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 [5:ch. 5]. It is not clear whether the results will hold for full Forsythe, which includes less restrictive intersection types.

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., Russel [4], Scheme (Lisp) [2, 8, 9, 12, 13], 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 [10, 16]. None of these languages or calculi is comparable to IA with respect to (imperative) expressive power. 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.

In conclusion, we believe that our work correctly captures the principles of Algol and that it continues the tradition of formalizing and exposing meta-properties of programming languages. Conversely, in order to determine whether a language is an extension of Algol or whether it belongs to a different class of programming languages, it suffices to check whether its calculus satisfies the above-mentioned properties or not. We conjecture that it is also possible to characterize other classes of languages (e.g., Scheme or ML) through the meta-properties of their calculi.

Acknowledgements. We appreciate comments by Robert Cartwright, Ian Mason, Amr Sabry, and John Gateley on an early draft.

References

1. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised Edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.
2. Crank, E. and M. Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, 1991, 233–245.
3. Daalen van, D. *The Language Theory of AUTOMATH.* Ph.D. Dissertation, Eindhoven University, 1980.
4. Demers, A. and J. Donahue. Making variables abstract: an equational theory for Russell. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983, 59–72.
5. Felleisen, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. dissertation, Indiana University, 1987.
6. Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming* **17**, 1991, 35–75. Preliminary version in: *Proc. 3rd European Symposium on Programming.* Neil Jones, Ed. Lecture Notes in Computer Science, 432. Springer Verlag, Berlin, 1990, 134–151.
7. Felleisen, M. and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.
8. Felleisen, M. and D.P. Friedman. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on*

Principles of Programming Languages, 1987, 314-325.

9. FELLEISEN, M. AND R. HIEB. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theor. Comput. Sci.* **102**, 1992.
10. GUZMÁN, J.C. AND P. HUDAK. Single-threaded polymorphic lambda-calculus. In *Proc. Symposium on Logic in Computer Science*, 1990, 333–345.
11. LENT, A. F. *The Category of Functors from State Shapes to Bottomless CPOs is Adequate for Block Structure.* Master's thesis, MIT, 1992.
12. MASON, I.A. AND C. TALCOTT. Equivalence in functional programming languages with effects. *Journal of Functional Programming* **1**(3), July 1991, 287–327. Preliminary version in: *Proc. International Conference on Automata, Languages and Programming*. Springer Lecture Notes in Computer Science, Vol. 372, Berlin, 1989, 574-588.
13. MASON, I.A. AND C. TALCOTT. Inferring the equivalence of functional programs that mutate data. *Theor. Comput. Sci.* **105**(2), 1992, 167–215. Preliminary version in: *Proc. Symposium on Logic in Computer Science*. Computer Society Press, Washington, D.C., 1989, 284–293.
14. NAUR, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM* **6**(1), 1963, 1–17.
15. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.
16. REDDY, U.S., V. SWARUP, AND E. IRELAND. Assignments for applicative languages. In *Proc. Conference on Functional Programming and Computer Architecture*. Lecture Notes in Computer Science, Vol. 523. Springer Verlag, Berlin, 1991, 192-214.
17. REYNOLDS, J.C. Replacing complexity with generality: The programming language Forsythe. Unpublished manuscript, Carnegie Mellon University, Computer Science Department, 1991.
18. REYNOLDS, J.C. The essence of Algol. In *Algorithmic Languages*, edited by de Bakker and van Vliet. North-Holland, Amsterdam, 1981, 345–372.
19. REYNOLDS, J.C. Preliminary Design of the Programming Language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Computer Science Department, 1988.
20. SØNDERGARD, H. AND P. SESTOFT. Referential transparency, definiteness and unfoldability. *Acta Informatica* **27**, 1990, 505–517.
21. WADLER, P. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990, 61–78.
22. WRIGHT, A. AND M. FELLEISEN. A syntactic approach to type soundness. Technical Report 160. Rice University, 1991. *Information and Computation*, 1992, to appear.

A Strong Normalization

In order to show that the orthogonal combination of two strongly-normalizing systems is strongly-normalizing, we would like a theorem similar to the following, found in van Daalen [3:80]:

Theorem A.1
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 the above 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 A.2. (*Strong Postponement*) $\mathbf{r}_1\mathbf{r}_2$ 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. Offhand, 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 Σ^*, 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 β, ρ, 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, namely, 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 which characterizes strongly-normalizing systems which prohibit arbitrary length reductions starting with a given term.

Definition A.3. (*Reachability*) 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 A.4
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.*

We begin by making some observations about reachability. We then prove the necessary lemmas for Theorem A.4, and finally the theorem itself.

Observation A.5

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

Observation A.6 *Bounded reduction length implies strong normalization.*

Proposition A.7 *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 A.5 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 A.8 *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. ∎

We are now ready to prove our main theorem.

Proof. (Theorem A.4) We show that $\mathbf{r}_1\mathbf{r}_2$ satisfies bounded reduction length, thus by Observation A.6 is strongly-normalizing. By Propositions A.7 and A.8 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 A.7) 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 $l \leq m + n \leq m' + n'$. ∎