A Staging Calculus and its Application to the Verification of Translators

(Preliminary Report)

Robert Muller*
Apple Computer
One Main Street
Cambridge, MA 02142
muller@cambridge.apple.com

Abstract

We develop a calculus in which the computation steps required to execute a computer program can be separated into discrete stages. The calculus, denoted λ_2, is embedded within the pure untyped λ-calculus. The main result of the paper is a characterization of sufficient conditions for confluence for terms in the calculus. The condition can be taken as a correctness criterion for translators that perform reductions in one stage leaving residual redexes over for subsequent computation stages. As an application of the theory, we verify the correctness of a macro expansion algorithm. The expansion algorithm is of some interest in its own right since it solves the problem of desired variable capture using only the familiar capture avoiding substitutions.

1 Introduction

The λ-calculus is widely used as a metalanguage for programming language semantics because most of the complexities of real programming languages can be modeled by relatively simple and well-understood aspects of the calculus. One important aspect of a programming language that does not have an obvious analog in the λ-calculus is the notion of *staging* — the fact that it is usually desirable to separate the computations required to execute a source program into discrete stages: usually *translation-time* and *run-time*. *Compilers*, *macro expanders* and *partial evaluators* are examples of translators that perform some computations statically and emit a *residual* program that will perform the remaining computations in a separate run-time stage.

In this paper we develop some syntactic theory for a calculus in which computation stages are represented explicitly. Although the formalism can account for an arbitrary number of stages, in this report we will restrict our attention to the two-stage case: compile-time and run-time. The calculus, denoted λ_2, is embedded within the pure untyped λ-calculus and was first motivated by the problem of verifying the correctness of macro expansion.

The main contribution of the paper is a characterization of sufficient conditions for confluence. The condition can be taken as a correctness criterion for translators that perform reductions in one stage leaving residual redexes over for subsequent stages when more input is available. As an application of the theory, we verify the correctness of a simple macro expansion algorithm.

It is not unusual to model translation-time computation steps by ordinary β-reduction. A translator $T \in \Lambda$ operates on a suitable representation $\ulcorner M \urcorner$ of a source program M and yields a representation $\ulcorner M' \urcorner$ of a residual program M'.

$$T \ulcorner M \urcorner \twoheadrightarrow_\beta \ulcorner M' \urcorner \tag{1}$$

In this paper we will restrict our attention to λ-calculus based source languages. The reader will recall the well-known fact that there is no $R \in \Lambda$ such that $\forall M \in \Lambda$, $RM =_\beta \ulcorner M \urcorner$ — representations of terms must be given *a priori*. Although a variety of representations would suffice, in this paper we will use Mogensen's representation [Mog92a]:

$$\begin{aligned} \ulcorner x \urcorner &\equiv \lambda abc.ax \\ \ulcorner (M\ N) \urcorner &\equiv \lambda abc.b \ulcorner M \urcorner \ulcorner N \urcorner \\ \ulcorner (\lambda x.M) \urcorner &\equiv \lambda abc.c(\lambda x. \ulcorner M \urcorner) \end{aligned}$$

The key idea of the λ_2-calculus is a notion of *representation* or *run-time reduction*. We define:

$$\ulcorner \beta \urcorner = \{(\ulcorner (\lambda x.M)N \urcorner, \ulcorner M[x := N] \urcorner) \mid M, N \in \Lambda\} \tag{2}$$

and consider the usual derived relations $\rightarrow_{\ulcorner \beta \urcorner}$, $\twoheadrightarrow_{\ulcorner \beta \urcorner}$ and $=_{\ulcorner \beta \urcorner}$. It is clear that although $=_{\ulcorner \beta \urcorner}$ identifies distinct β-normal-forms, $=_{\ulcorner \beta \urcorner}$ is nevertheless a consistent theory since $\ulcorner \beta \urcorner$ is a "mirror image" of β.

λ_2 is obtained by combining these notions of reduction:

$$\beta_2 = \beta \cup \ulcorner \beta \urcorner \tag{3}$$

In λ_2, one thinks of a representation $\ulcorner (\lambda x.M)N \urcorner$ as an "active datatype" that can either be reduced (that is, to $\ulcorner M[x := N] \urcorner$) or taken apart (that is, to $\ulcorner \lambda x.M \urcorner$ and $\ulcorner N \urcorner$). The activity reflects what the represented code could do in subsequent computation stages — in this case, run-time.

*This work was partially supported by the Defense Advanced Research Projects Agency under grant number F19628-92-C-0113.

POPL 94- 1/94, Portland Oregon,USA

One of the key problems taken up in this paper is to determine the weakest conditions sufficient to ensure that the translation-time operations on the data type do not interfere with its intended run-time behavior. We take confluence to be an appropriate notion of non-interference.

For unrestricted λ-terms the relation β_2 is not confluent. For example,

$$\ulcorner(\lambda u.v)w\urcorner x \tag{4}$$

where u, v, w and x are variables, has two distinct β_2-normal-forms: $\lambda bc.xv$ and $\lambda bc.b\ulcorner\lambda u.v\urcorner\ulcorner w\urcorner$. In order to rule-out such ill-formed terms we impose a simple type discipline. We define the *representation types* by

$$\tau ::= d \mid \tau \rightarrow \tau$$

We write $M : d$ if there exists an N such that $M =_\beta \ulcorner N\urcorner$ and $M : \sigma \rightarrow \tau$ if M maps every term of type σ to one of type τ. Intuitively, if there exists a τ such that $M : \tau$ then M has a reasonable shape.

Unfortunately, well-typedness does not suffice to ensure confluence under β_2 since well-typed terms may not respect the run-time equational theory. For example, taking I, K and S to be the usual combinators, the term:

$$\ulcorner I\ I\urcorner(\lambda x.\ulcorner I\urcorner)(\lambda xy.\ulcorner K\urcorner)(\lambda x.\ulcorner S\urcorner) \tag{5}$$

is of type d but has two distinct β_2-normal-forms: $\ulcorner K\urcorner$ and $\ulcorner S\urcorner$. Intuitively, the problem is that reducing directly to a β-normal-form leads to a residual program $\ulcorner K\urcorner$ whose behavior differs from that obtained by first reducing the run-time redex $\ulcorner II\urcorner$ and then reducing to β-normal-form. The key restriction on terms, *well-behavedness*, is intended to be the weakest condition that rules-out such terms.

That well-typed and well-behaved terms are not built up inductively from well-typed and well behaved subterms will prove to be a key problem for proving confluence. Our proof will rely on reasoning about β-normal-forms.

As an application of the theory, we will develop a macro expansion algorithm for a simple λ-calculus-based language and verify its correctness. This application is of interest because one would like to think of macros as inessential abstractions of patterns of syntax. One way to characterize this is to say that their expansion can be interleaved with term reduction without affecting the outcome. It is well-known that this property often fails for macro systems that introduce binding occurrences of variables. As an example, consider the simple *Or* macro:

$$\text{Or } P\ Q \stackrel{\text{df}}{\equiv} (\lambda x.\text{if } x \text{ then } x \text{ else } Q)P, \tag{6}$$

An unintended variable capture can occur on expanding

$$(\text{Or } M_1\ M_2)$$

whenever $x \in FV(M_2)$ and it is then easy to see that expansion cannot be interleaved with term reduction. A concrete example is:

$$(\lambda x.(\text{Or False } x))\ \text{True}$$

in which (naive) expansion of the macro followed by β-reduction results in *False* while performing the β-redex first results in *True*.

Our expansion algorithm is of interest not only because it is quite explicit about term representation but also because it solves the aforementioned problems using only the familiar capture avoiding substitutions.

The remainder of this paper is organized as follows. In Section 2 we introduce well-behaved terms and illustrate some of their properties with examples. In Section 3 we prove the confluence theorem. In Section 4 we develop the macro expansion algorithm and prove its correctness using the main theorem. In Section 5 we compare the present work with related work, emphasizing the connection to [Bar91, Mog92b, Wan93] and to [Gri88]. Section 6 sketches some future lines of research. Detailed proofs can be found in [Mul93].

2 Well-Behaved Terms

2.1 Preliminaries

Let $Var = \{x_1, x_2, \ldots\}$ be a countable set of variables. We use the symbol Λ for the set of terms:

$$\Lambda ::= x \mid \lambda x.\Lambda \mid \Lambda\Lambda$$

We use $M \equiv N$ to denote α-congruent terms. Let

$$R = \{(M, N) \mid \text{conditions}\}$$

be a reduction relation. We use $R\text{-nf}(M)$ to denote the R-normal-form of M and $\rightarrow_R$, $\twoheadrightarrow_R$ and $=_R$ to denote the one-step, many-step and congruence relations (resp.) generated by R. Let P be a property of terms. We use the notation $P_R(M)$ to mean that P holds for all M under R-reduction. Contexts $C[\cdot]$ are defined in the usual way. We will often use the standard combinators $I \equiv \lambda x.x$, $K \equiv \lambda xy.x$, $S \equiv \lambda xyz.xz(yz)$ and $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ for illustration.

We use $\ulcorner M\urcorner$ to denote Mogensen's representation of M which is presented in Section 1. We note that $\ulcorner\cdot\urcorner$ is an injective higher-order abstract syntax, that $\ulcorner M\urcorner$ is a β-normal-form and that

$$\text{FV}(\ulcorner M\urcorner) = \text{FV}(M),$$

where $\text{FV}(M)$ denotes the free variables of M.

The notion of reduction $\ulcorner\beta\urcorner$ is as defined in (2). From the remarks above we have the following properties of $\ulcorner\beta\urcorner$.

Lemma 1

1. *The relation $\ulcorner\beta\urcorner$ is Church-Rosser.*
2. *For M, $N \in \Lambda$, $M =_\beta N \Leftrightarrow \ulcorner M\urcorner =_{\ulcorner\beta\urcorner} \ulcorner N\urcorner$.*

The main object of interest in this paper is β_2 as defined in (3). As we illustrated in Section 1, β_2 is not confluent on unrestricted terms.

2.2 Representation Types

In this section we develop a type discipline to rule out ill-formed terms. The type expressions are given by:

$$\tau ::= d \mid \tau \rightarrow \tau$$

where the type constant d denotes *dynamic*. We will use the symbol τ to denote an arbitrary representation type and occasionally τ_M to denote any type expression assignable to term M.

The definition of well-typed terms is given in terms of β-convertibility.

Definition 1 (Well-Typed) For $M \in \Lambda$ we define $M : \tau$ as follows:

1. $M : d$ iff $\exists N \in \Lambda$, such that $M =_\beta \ulcorner N \urcorner$.
2. $M : \sigma \to \tau$ iff M is not of type d and $\forall N : \sigma$, $MN : \tau$.

We say M is *well-typed* iff there exists a τ such that $M : \tau$.

Given M and τ, it is obviously undecidable whether $M : \tau$.

2.2.1 Examples

In Figure 1 we present several examples to illustrate properties of well-typed terms. In (7), $\lambda x.x$ has type $\tau \to \tau$ for any type τ. This, reflects the usual polymorphism of λ-terms. (8) is an *expander function* for a *let* macro taken from the macro system presented in Section 4. The expansion function produces a representation of an application when applied to appropriate representations of the parts of a use of the macro. (9) is the ill-formed case from (4) that motivated the type system. (10) illustrates that well-typed terms can have ill-typed subterms. (11) is the well-typed but badly-behaved case from (5).

We wish to stress the importance of the restriction "is not of type d" in the definition of higher type. The restriction is required to preserve important properties of the calculus. Without it, the subject reduction property fails — a property on which our confluence proof depends. Consider the term $\ulcorner \lambda x.y \urcorner \equiv \lambda abc.c(\lambda x.\ulcorner y \urcorner)$. Without the restriction it is easy to see that the term has the following incompatible types:

$$\ulcorner \lambda x.y \urcorner : \begin{cases} d, \\ \tau_a \to \tau_b \to ((\tau_x \to d) \to \tau) \to \tau \end{cases} \qquad (12)$$

It is straightforward to construct a similar term with $\ulcorner \beta \urcorner$-redexes (e.g., $\ulcorner (\lambda x.y)(\lambda x.y) \urcorner$) whose higher type is not preserved under $\ulcorner \beta \urcorner$-reduction.

2.2.2 Properties of Well-Typed Terms

We briefly consider some properties of well-typed terms. Strong normalization obviously fails for both β and β_2 (e.g., $K \ulcorner \Omega \urcorner \Omega$). The same example also shows that M will not in general have a β_2-normal-form. However, well-typed terms do have the weak-normalization property.

Lemma 2 (Weak Normalization) If $M : \tau$ then M has a β-normal-form.

2.3 Well-Behavedness

As we have stressed, well-typedness is not a strong enough condition to guarantee the integrity of staged execution. The example in (5) has two distinct β_2-normal-forms: $\ulcorner K \urcorner$ and $\ulcorner S \urcorner$. We now define an additional criterion, *well behavedness*, with the intention of ruling-out such terms.

We begin by reviewing the notion of a *self-interpreter* (or *enumerator*) first introduced by Kleene [Kle36].

Theorem 1 (Kleene)

$$\exists \mathbf{E} \in \Lambda, \text{ s.t. } \forall M \in \Lambda^0, \mathbf{E} \ulcorner M \urcorner =_\beta M.$$

For Mogensen's representation, such an enumerator is quite simple:

$$\begin{aligned} \mathbf{E} \equiv Y(\lambda em.m(\lambda x.x) \\ (\lambda mn.(em)(en)) \\ (\lambda m.\lambda v.e(mv))) \end{aligned}$$

We will make use of the enumerator in defining an equivalence relation on terms of base type. Two terms will be considered equivalent if and only if their β-normal-forms denote equivalent run-time code.

$$M \approx N \text{ iff } M : d, N : d \text{ and } \mathbf{E}M =_\beta \mathbf{E}N.$$

From lemma 1 (part 2) and the definition we have:

Lemma 3 *For all M, N, if $\ulcorner M \urcorner =_{\ulcorner \beta \urcorner} \ulcorner N \urcorner$ then $M \approx N$.*

The converse obviously does not hold.

We can now give the key definition.

Definition 2 (Well-Behaved)

1. M is *well-behaved at d* iff $M : d$ and $\forall C[\cdot]$, $N_1 : d$ s.t. $M \equiv C[N_1]$, $\forall N_2$, s.t. $N_1 \approx N_2$, $C[N_1] \approx C[N_2]$.
2. M is *well-behaved at $\sigma \to \tau$* iff $M : \sigma \to \tau$ and for all well-behaved $N : \sigma$, $MN : \tau$ is well-behaved.

M is *well-behaved* iff M is well-behaved at τ for all τ such that $M : \tau$.

The intuition behind *1.* is that given $C[N_1] : d$, a translator ought to be able to replace the code fragment $N_1 : d$ with any equivalent (and hopefully more efficient) code fragment $N_2 : d$. *2.* reflects the natural higher representation type of any translator.

Examples

1. $\lambda x.x$ is well-behaved.
2. $\ulcorner \lambda x.x \urcorner$ is well-behaved.
3. $(\lambda mn.\lambda abc.b(\lambda abc.c\lambda x.(n \ulcorner x \urcorner))m)$ is well-behaved.
4. $K \ulcorner I \urcorner (\ulcorner (\lambda u.v) w \urcorner x)$ is well-behaved.
5. $\ulcorner I\ I \urcorner (\lambda m.\ulcorner I \urcorner)(\lambda mn.\ulcorner K \urcorner)(\lambda m.\ulcorner S \urcorner)$ is not well behaved.

It is obviously undecidable whether a given term M is well behaved.

Lemma 4 (Subject Reduction) If $M : \tau$ is well-behaved and $M \twoheadrightarrow_{\beta_2} N$ then $N : \tau$.

$$\lambda x.x \;:\; \tau \to \tau \tag{7}$$
$$(\lambda mn.\lambda abc.b(\lambda abc.c\lambda x.(n\ \ulcorner x \urcorner))m) \;:\; d \to (d \to d) \to d \tag{8}$$
$$\ulcorner(\lambda u.v)w\urcorner x \quad \text{has no type} \tag{9}$$
$$K\ulcorner I\urcorner(\ulcorner(\lambda u.v)w\urcorner x) \;:\; d \tag{10}$$
$$\ulcorner I\ I\urcorner(\lambda m.\ulcorner I\urcorner)(\lambda mn.\ulcorner K\urcorner)(\lambda m.\ulcorner S\urcorner) \;:\; d \tag{11}$$

Figure 1: Example Terms and their Types

3 Confluence of Well-Behaved Terms

We now turn to the main result of the paper: confluence for well-behaved terms under β_2-reduction. Unfortunately, most of the usual methods for proving confluence of rewriting systems are of no help for the λ_2-calculus. The general results of Klop [Klo80] for regular term rewriting systems do not apply because β_2 is not regular: β and $\ulcorner\beta\urcorner$ interfere with one-another. The Hindley-Rosen lemma [Bar84], (i.e., the union of commutative CR reduction relations is CR) doesn't apply because, although both β and β_2 are CR, $\ulcorner\beta\urcorner$ does not commute with β on well-behaved terms (for example, $K\ulcorner M\urcorner(\ulcorner(\lambda u.v)w\urcorner x)$.) Similar problems arise with development based proofs.

The key problem is that the relations $M : \tau$ and $M : \tau$ *is well-behaved* are not defined in such a way that there is an obvious basis for reasoning by induction on the structure of terms or on the lengths of reduction sequences. One important consequence of the definitions is that the relation $\ulcorner\beta\urcorner$ is not substitutive. That is, for well-behaved $M : \tau$,

$$M \twoheadrightarrow_{\ulcorner\beta\urcorner} M' \not\Rightarrow M[x := N] \twoheadrightarrow_{\ulcorner\beta\urcorner} M'[x := N].$$

The definition of well-behavedness is given in terms of β-normal-forms and so our proof of confluence relies on reasoning about β-normal-forms. We will require the following two lemmas.

Lemma 5 *If $M_0 : \tau$ is well-behaved and $M_0 =_{\beta_2} M_n$ then M_n is well-behaved.*

Proof: The proof is by induction on τ and on why $M_0 =_{\beta_2} M_n$. ■

Lemma 6 *If $M_0 : \tau$ is well-behaved and $M_0 =_{\beta_2} M_n$ then β-nf$(M_0) =_{\ulcorner\beta\urcorner} \beta$-nf$(M_n)$.*

Proof: (Sketch) The proof is by induction on τ and on why $M_0 =_{\beta_2} M_n$. We briefly consider the base case. Let $\tau = d$, and consider why $M_0 =_{\beta_2} M_n$. The non-trivial case is $M_0 =_{\beta_2} M_n$ because $M_0 \twoheadrightarrow_{\beta_2} M_n$. We proceed by induction on n. For $n = 1$ let $C[\cdot]$, N_0 and N_1 be such that $M_0 \equiv C[N_0]$ and $M_1 \equiv C[N_1]$ where $(N_0, N_1) \in \beta_2$. If $(N_0, N_1) \in \beta$, then β-nf(M_0) and β-nf(M_1) are identical and the result is immediate. If $(N_0, N_1) \in \ulcorner\beta\urcorner$, then since M_0 is well-behaved, and $N_0 \approx N_1$, $M_0 \approx M_1$. Let $\ulcorner M_i'\urcorner \equiv \beta$-nf$(M_i)$. Then

$$\begin{aligned} M_0 \approx M_1 \quad &\text{iff} \quad \mathbf{E}M_0 =_\beta \mathbf{E}M_1 \\ &\text{iff} \quad \mathbf{E}\ulcorner M_0'\urcorner =_\beta \mathbf{E}\ulcorner M_1'\urcorner \\ &\text{iff} \quad M_0' =_\beta M_1' \\ &\text{iff} \quad \beta\text{-nf}(M_0) =_{\ulcorner\beta\urcorner} \beta\text{-nf}(M_1). \end{aligned}$$

For $n > 1$, $M_0 \twoheadrightarrow_{\beta_2} M_{n-1} \to_{\beta_2} M_n$ and by the induction hypothesis β-nf$(M_0) =_{\ulcorner\beta\urcorner} \beta$-nf$(M_{n-1})$. If $M_{n-1} \to_\beta M_n$ the result follows from the induction hypothesis. If

$$M_{n-1} \to_{\ulcorner\beta\urcorner} M_n$$

the result follows by the same reasoning as in the base case noting that M_{n-1} is of type d by lemma 4 and well-behaved by lemma 5. ■

Theorem 2 (Confluence) If $M : \tau$ is well behaved, then $\forall M_1, M_2$ such that $M \twoheadrightarrow_{\beta_2} M_1$ and $M \twoheadrightarrow_{\beta_2} M_2$ there exists an M_3 such that $M_1 \twoheadrightarrow_{\beta_2} M_3$ and $M_2 \twoheadrightarrow_{\beta_2} M_3$.

Proof: By lemma 2, M, M_1 and M_2 all have β-normal-forms which, by lemma 6, are $\ulcorner\beta\urcorner$-convertible. Confluence then follows by two applications of lemma 1. ■

4 Well-Behaved Macro Expansion

In this section we present a simple λ-calculus based language with syntax extensions and an algorithm for expanding instances of defined notation. It seems natural to view such an algorithm as being correct if expansion of defined notation commutes with reduction on terms. As we illustrated in Section 1, naive expansion fails to satisfy this criterion because unintended captures of free variables can occur during naive expansion. This problem has been taken up many times before [KFFD86, Gri88, CR91, BA92] and [Car93]. We will compare our approach with some of these in Section 5.

In most respects our approach will be the usual one for macro expansion: instead of the translation process depicted in (1) in which the front-end of the translator has produced a β-normal-form $\ulcorner M\urcorner$, the front end produces a term M_0 in which definitions of macros are represented by expansion functions and uses of macros are represented by applications of the expansion functions to representations of the "parts" of the macro use. The translation process can then be seen as:

$$T\ M_0 \twoheadrightarrow_\beta T\ \ulcorner M\urcorner \twoheadrightarrow_\beta \ulcorner M'\urcorner,$$

where all macro calls in M_0 can be fully expanded by β-reduction yielding the β-normal-form $\ulcorner M\urcorner$ and the compiler then produces the representation of the object program $\ulcorner M'\urcorner$.

4.1 Preliminaries

The grammar for our language $\mathcal{L}$ is defined in Figure 2. The symbol x denotes a set $\{x_1, x_2, \ldots\}$ of object variables and v denotes a disjoint set of pattern variables $\{v_1, v_2, \ldots\}$ together with the special symbol ϵ which denotes *empty* (i.e.,

$$\begin{aligned}
\Lambda &::= x \mid \Lambda\,\Lambda \mid \lambda z.\Lambda \mid \text{let-syntax } D \text{ in } \Lambda \mid U \mid v \\
D &::= v\ v^{v'}\ \ldots \stackrel{\text{df}}{\equiv} \Lambda \\
U &::= v\ \Lambda\ \ldots
\end{aligned}$$

Figure 2: Grammar for $\mathcal{L}$.

no variable). The symbol z denotes either an object variable or a pattern variable. D denotes a syntax *definition* and U denotes a *use* of the defined notation. The ellipses "$\Lambda\ldots$" denotes a sequence of Λ's. The language provides for lexically nested macro definitions. Thus, a use U will be meaningful only as a subterm of Λ in *let-syntax* D *in* Λ. Similarly, a pattern variable v is meaningful only in a macro definition.[1]

Following Griffin, we use superscripts $v^{v'}$ to denote binding patterns within definitions. For example, in:

$$\text{let } v_1 = v_2 \text{ in } v_3^{v_1} \stackrel{\text{df}}{\equiv} (\lambda v_1.v_3)v_2. \tag{13}$$

the superscript v_1 in $v_3^{v_1}$ indicates that free occurrences of the variable that v_1 is bound to that occur in the expression that v_3 is bound to will ultimately be bound by the corresponding variable on the right-hand side. We refer to such superscripts as *scope* variables. While our language allows only a single scope variable, it is easy to extend the algorithm to sets of variables $v^{\{v_1,\ldots,v_k\}}$.)

We are making a number of other simplifying assumptions that set aside problems that we are not attempting to solve here. For example, constant symbols and/or literal strings are important for making macros readable, however, we have not included them here (and use object and pattern variables informally as constants) to keep the equations simple. Similarly, we are not addressing the problems of parsing the concrete syntax (but see [Car93, Mul89]) or problems of programmers specifying compile-time computations over representations — backquote.

The algorithm will translate terms relative to an environment $\eta ::= \varepsilon \mid \eta[A/v]$ that binds pattern variables v to values A. Environments have the following behavior:

$$\begin{aligned}
\varepsilon(v) &\equiv \epsilon \\
\eta[A/v'](v) &\equiv \begin{cases} A & v \equiv v', \\ \eta(v) & \text{otherwise.} \end{cases}
\end{aligned}$$

We will find it convenient to retain to two kinds of values in η:

$$A ::= (F_k, (i_1, j_1), \ldots, (i_k, j_k)) \mid \ulcorner v \urcorner,$$

where F_k denotes a k-ary expansion function and i, j are integers that are included to facilitate construction of the application that will expand a use of the macro.

Finally, we will find it convenient to adopt the following two conventions. First, for all v^{v_i} in which $v_i \equiv \epsilon$, $i = 0$. Intuitively, i is the index of a variable so the convention says that the empty variable has no index. Second, we use the non-standard notation $\lambda M_i.M_j$. In such contexts M_i denotes either a (non-empty) variable x, or, if $i = 0$, then $\lambda M_i.M_j$ denotes M_j. These conventions will simplify the key equations of the algorithm.

[1] Strictly speaking, we are also using them as identifiers of macros in uses. This pun allows us to avoid introducing new syntactic categories and keep the equations simple.

4.2 The Expansion Algorithm

The translation algorithm is given in Figure 3. From the definition it is easy to see that if M has no macro uses then $[\![M]\!]\eta$ is in β-normal-form — (14) through (16) yield a Mogensen representation. The translation of a *let-syntax* form (17) results in the translation of M in the environment obtained by translating the definition D.

The translation of a definition (18) extends the environment by binding the leftmost symbol of the definition to the appropriate expansion function together with pairs of indices (i_{j_l}, j_l), $1 \le l \le k$. The body of the expansion function is determined by the translation of the right-hand side of the definition in an environment in which the k parameters of the function are bound to their scope variables. (These will eventually be looked up in (20).) The pairs (i_{j_l}, j_l), are packaged-up with the expansion function so that they can be used in translating a use of the defined notation. The index j_l gives the position in a use of the lth argument to which the expansion function will be applied. If the index $i_{j_l} = 0$ then the j_lth term in the use is not within the scope of any variable in the macro use. If $i_{j_l} \neq 0$ then the j_lth term is in the scope of the variable at index i_{j_l}.

Turning to the translation of a macro use (19), the algorithm produces an application of the k-ary expansion function associated with the leftmost symbol v to k arguments. For $1 \le l \le k$, the lth argument is written as an abstraction $\lambda M_{i_{j_l}}.[\![M_{j_l}]\!]\eta$ where the "formal parameter" $M_{i_{j_l}}$ is determined by the index as described above. When $i_{j_l} \neq 0$, M_{j_l} is in the scope of variable $M_{i_{j_l}}$ and the abstraction will ultimately be applied to the representation of the appropriate variable (thus performing the capture). When $i_{j_l} = 0$, no capture was specified in the definition and by our convention $\lambda M_{i_{j_l}}.[\![M_{j_l}]\!]\eta$ is simply $[\![M_{j_l}]\!]\eta$.

Finally, the last clause (20) gives the translation of a pattern variable v. If $\eta(v) = v'$, for some $v' \neq \epsilon$ then a capture was specified for v in the corresponding definition and v is applied to the representation of v'. If $\eta(v) = \epsilon$ then no capture was specified and no application is generated.

As we will show, the algorithm constructs a representation of the specified binding pattern using only abstraction, application and the familiar capture avoiding substitution. Recall that the key clause is:

$$(\lambda x.M)[y := N] \equiv (\lambda z.M[x := z][y := N]) \tag{21}$$

where $x \not\equiv y$ and z is unique. The basic technique was first suggested to us by Pat O'Keefe [O'K92].

$$[\![x]\!]\eta = \lambda abc.ax \quad (14)$$

$$[\![(M\ N)]\!]\eta = \lambda abc.b([\![M]\!]\eta)([\![N]\!]\eta) \quad (15)$$

$$[\![(\lambda z.M)]\!]\eta = \lambda abc.c(\lambda z.[\![M]\!]\eta) \quad (16)$$

$$[\![\text{let-syntax } D \text{ in } M]\!]\eta = [\![M]\!]([\![D]\!]\eta) \quad (17)$$

$$[\![v\ v_1^{v_{i_1}}\ \ldots\ v_n^{v_{i_n}} \stackrel{\text{df}}{\equiv} M]\!]\eta = \eta[(\lambda v_{j_1} \ldots v_{j_k}.[\![M]\!](\eta[\ulcorner v_{i_{j_1}} \urcorner/v_{j_1}] \ldots [\ulcorner v_{i_{j_k}} \urcorner/v_{j_k}]), (i_{j_1}, j_1), \ldots, (i_{j_k}, j_k))/v] \quad (18)$$

$$[\![v\ M_1\ \ldots\ M_n]\!]\eta = (F_k\ \lambda M_{i_{j_1}}.[\![M_{j_1}]\!]\eta\ \ldots\ \lambda M_{i_{j_k}}.[\![M_{j_k}]\!]\eta) \quad (19)$$

$$\text{where } \eta(v) = \langle F_k, (i_{j_1}, j_1), \ldots, (i_{j_k}, j_k)\rangle$$

$$[\![v]\!]\eta = \text{if } \eta(v) \equiv \ulcorner v' \urcorner \text{ then } v\ulcorner v' \urcorner \text{ else } v \quad (20)$$

Figure 3: The Macro Transcription Algorithm

4.3 Example

The following example illustrates the algorithm's method of effecting desired captures while avoiding undesired captures. In the example we will feel free to use M, N, ..., and even x as pattern variables and we will continue to be rather informal with constants (using "=" and "in" for example as term constants.) We will also assume the existence of a conditional $M \rightarrow M, M$.

$$\begin{aligned} D_1 &\equiv \text{let } x = M \text{ in } N^x \stackrel{\text{df}}{\equiv} (\lambda x.N)M \\ D_2 &\equiv \text{or } P\ Q \stackrel{\text{df}}{\equiv} (\text{let } x = P \text{ in } (x \rightarrow x, Q)) \\ U &\equiv (\text{or } (I\ I)\ x) \\ M &\equiv \text{let-syntax } D_1 \text{ in} \\ &\qquad \text{let-syntax } D_2 \text{ in } U \end{aligned}$$

Let $let' \equiv (\lambda mn.\lambda abc.b(\lambda abc.c\lambda x.(n\ \ulcorner x \urcorner))m)$. We compute:

$$[\![D_1]\!]\epsilon = \epsilon[\langle let', (0,3), (1,5)\rangle/\text{let}].$$

Let $or' \equiv (\lambda pq.(let'\ p\ (\lambda x.x\ \ulcorner \rightarrow \urcorner\ x, q)))$. Then

$$[\![D_2]\!]([\![D_1]\!]\epsilon) = ([\![D_1]\!]\epsilon)[\langle or', (0,1), (0,2)\rangle/\text{or}].$$

Note that the application of n to $\ulcorner x \urcorner$ in let' together with the abstraction over x in or' jointly effect the capture specified in the definition of D_1.

The translation and expansion then proceed as in Figure 4. Note in (22) that the normal replacement rule for capture avoiding substitution (21) prevents the capture of x in the call by the binding instance of x introduced in the definition of D_2. Also note that in (23), the usual α-conversion again takes place converting $\lambda x.(n\ \ulcorner x \urcorner)$ to $\lambda x''.(n\ \ulcorner x'' \urcorner)$ — *a fortiori*, in Mogensen's higher-order abstract syntax, α-conversion is performed across representation levels. Finally, the book-keeping β-reduction is performed in (24) that effects the capture specified in the definition of D_1.

4.4 Correctness

Our correctness criterion is that macro expansion should commute with β-reduction on $\mathcal{L}$-terms.

Theorem 3 (Correctness of Macro Expansion) $[\![\cdot]\!]$ is correct.

Proof: (Sketch) By the confluence theorem, it suffices to show that for all $M \in \mathcal{L}$, $[\![M]\!]\epsilon$ is well-behaved. We first show that for all $M \in \mathcal{L}$, $[\![M]\!]\epsilon : d$. We then show by induction on $[\![M]\!]\epsilon$ that for all $C[\cdot]$, $N_1 : d$ such that $C[N_1] = [\![M]\!]\epsilon$, and for all $N_2 : d$ such that $N_1 \approx N_2$, that $C[N_1] \approx C[N_2]$. ■

5 Related Work

The work described here is closely related to the syntactic approaches to self-interpretation studied in [Bar91] and self-interpretation and partial evaluation developed in [Mog92b] and [Wan93]. Following Gomard [Gom90], the latter two papers present type systems that compute a *binding-time analysis* as a preprocessing phase of a partial evaluator. The analysis yields a set of constraints on an annotated type inference tree. Those redexes with static annotations can be reduced statically while those with dynamic annotations cause the partial evaluator to emit residual code. The main result of [Wan93] is a soundness theorem verifying the correctness of the staging transformation for well-typed (well-annotated) terms.

Let us outline some of the differences between this approach and that presented in this paper. First, setting aside well-behavedness, the intention behind the type system developed here is that it give the weakest conditions that ensure that terms are sufficiently well-structured. Over and above the requirement that terms be well-typed, we have the additional requirement that ensures that a term is not only well-structured but that it also respects the run-time equational theory. As we have noted, this property is also undecidable.

We have not considered recursive types in part because our confluence result relies on weak normalization. It may be possible to obtain a confluence proof using logical relations, however, this would require a structural type system. Unfortunately, the natural axiom would be $\vdash \ulcorner m \urcorner : d$ which is obviously much weaker than our base type condition.

The application of the theory to macro expansion is closely related to (and to some extent was inspired by) Griffin's work [Gri88]. To the best of our knowledge, Griffin was the first to suggest commutativity as a correctness criterion for macro expansion. Griffin's framework was expressed in terms of the simply-typed λ-calculus extended with pairs. We share the use of higher-order abstract syntax to prevent unintended captures of free variables. One technical differ-

$$
\begin{array}{lcll}
[\![M]\!]\epsilon & = & [\![(\text{or } (I\ I)\ x)]\!]([\![D_2]\!]([\![D_1]\!]\epsilon)) & \\
 & = & ((\lambda pq.(let'\ p\ (\lambda x.x\ \ulcorner\rightarrow\urcorner\ x, q)))\ \ulcorner(I\ I)\urcorner\ \ulcorner x\urcorner) & \\
 & \twoheadrightarrow_\beta & (let'\ \ulcorner(I\ I)\urcorner\ (\lambda x'.x'\ \ulcorner\rightarrow\urcorner\ x', \ulcorner x\urcorner)) & (22) \\
 & \equiv & ((\lambda mn.\lambda abc.b(\lambda abc.c\lambda x.(n\ \ulcorner x\urcorner))m)\ \ulcorner(I\ I)\urcorner\ (\lambda x'.x'\ \ulcorner\rightarrow\urcorner\ x', \ulcorner x\urcorner)) & \\
 & \twoheadrightarrow_\beta & \lambda abc.b(\lambda abc.c\lambda x''.((\lambda x'.x'\ \ulcorner\rightarrow\urcorner\ x', \ulcorner x\urcorner)\ \ulcorner x''\urcorner))\ \ulcorner(I\ I)\urcorner & (23) \\
 & \rightarrow_\beta & \lambda abc.b(\lambda abc.c\lambda x''.\ulcorner x''\urcorner\ \ulcorner\rightarrow\urcorner\ \ulcorner x''\urcorner, \ulcorner x\urcorner)\ \ulcorner(I\ I)\urcorner & (24) \\
 & \equiv & \ulcorner(\lambda x''.x''\ \rightarrow\ x'', x)\ (I\ I)\urcorner &
\end{array}
$$

Figure 4: A Sample Macro Expansion

ence is that we have used explicit representations of terms whereas Griffin has adopted the representation framework from [HHP87] in which the source language is represented by adding an appropriate set of base types and constants. Another technical difference between our approaches is that Griffin used capture *permitting* substitutions to effect intended captures whereas we have used what we believe is a somewhat more direct method.

While Griffin's work is in many respects more general than ours, the present work does contribute a somewhat more general correctness criterion. Griffin develops an expansion algorithm $\mathcal{F}_\epsilon^\sigma$ and proves that under the algorithm, expansion of notational definitions commutes with reduction on terms. He then takes the algorithm as a correctness criterion — any other algorithm is correct if it agrees with $\mathcal{F}_\epsilon^\sigma$ on all inputs. Our criterion — well-behavedness — is independent of any particular expansion algorithm. Rather, it is directly tied to the notion of commutativity. Well-behavedness can therefore be used to prove the correctness of a larger class of macro expanders. For example, Griffin's criterion would exclude any expander that attempted to perform β-reduction statically.

The notion of *hygienic* [KFFD86] macro expansion has been widely adopted in the Scheme community as a correctness criterion for macro expansion. An expansion algorithm is said to be hygienic if it respects the following *hygiene condition* [KFFD86]:

> "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated in the same transcription step."

The classical hygienic expansion algorithm satisfies this condition by repeatedly α-converting variables during each transcription step.

While the approach taken in the algorithm of the preceding section using capture avoiding substitution has a superficial correspondence with the repeated α-conversion of hygienic expansion, we emphasize that the approaches are actually quite different.

Hygienic expansion is usually used in the context of S-expression based dialects of LISP. In these dialects, program fragments have no syntactical interpretation until all enclosing macro calls have been fully expanded. Thus, it is not clear how one can sensibly view macro expansion as being interleaved with run-time reduction.

Its also worth noting the connection between this property of S-expression LISP and the apparent gap in the definition of the hygiene condition — while the condition prevents capture of free variable occurrences generated in one transcription step by binding occurrences generated in another step, it makes no reference to variable occurrences generated in *different* binding contexts in the *same* transcription step. Let us consider a simple example.

```
let-syntax (mac a (b c d)) => (b c a) in
  ((lambda (x) (mac x (lambda (x) e))) y)
```

Expanding the macro (hygienically) first, we get

```
-->_Macro ((lambda (x) (lambda (x) x)) y)
-->_Beta (lambda (x) x)
```

But performing the β-reduction first it appears that we would get:

```
-->_Beta (mac y (lambda (x) e))
-->_Macro (lambda (x) y)
```

However, in S-expression LISP, the occurrence of the symbol x within the macro call is not taken as a variable and thus it is not bound by the left-most binding instance of x and both orders produce the same result. So the correctness of hygienic expansion is tied in some sense to this particular property of S-expression LISP.

While S-expressions provide for great expressive power for macro *writers* in terms of picking apart pieces of syntax, unfortunately, it leaves the macro *user* with only fairly weak assurances that their intended binding patterns will be preserved during expansion of their notational abbreviations. We believe that the classical S-expression syntactic structure is insufficient to ensure under reasonable conditions that macro expansion does not interfere with ordinary reduction in the sense that we have emphasized.

6 Future Work

Our definition of well-behavedness requires that a term M be well-behaved at τ for all τ such that $M : \tau$. It would be desirable if the well-behavedness of a polymorphic term could be inferred by its well-behavedness at a particular type along the lines of [Abr86]. We would like to come to a better understanding of the connection between the type system presented here and the two-level type system of [NN88]. We would also like to better understand the notion of staged reduction with other representation schemes such as that

presented in [BB92]. We hope to apply the current framework to the verification of Mogensen's self-applicable partial evaluator. Finally, it would be interesting to consider whether the equality condition could be weakened to allow a translator somewhat wider leeway in substitution of terms.

Acknowledgements

The author would like to thank Mitch Wand and the members of his semantics seminar for their valuable feedback. The author would also like to thank Tom Cheatham for his insight and support. Thanks to Pat O'Keefe for many long conversations about macros and valuable feedback on an earlier draft of this paper. Thanks to Allyn Dimock and Neil Jones for directing me to Mogensen's work.

References

[Abr86] S. Abramsky. Strictness analysis and polymorphic invariance. In *Programs as Data Objects (H. Ganzinger and N. Jones editors)*, pages 1–23. Springer-Verlag LNCS Vol. 217, 1986.

[BA92] A. Bove and L. Arbilla. A confluent calculus of macro expansion and evaluation. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 278–287, 1992.

[Bar84] H. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North Holland Press, Amsterdam, 1984.

[Bar91] H. Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1 (2):229–234, 1991.

[BB92] A. Berarducci and C. Bohm. A self-interpreter of lambda calculus having a normal form. In *Proceedings of 1992 Conference of the European Association for Computer Science Logic*, 1992.

[Car93] L. Cardelli. An implementation of $F_{<:}$. Technical Report SRC Research Report 97, Digital Equipment Corporation, 1993.

[CR91] W. Clinger and J. Rees. Macros that work. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 282–287, 1990.

[Gri88] T. Griffin. Notational definition — a formal account. In *Third Annual Symposium on Logic in Computer Science*, pages 372–383, 1988.

[HHP87] R. Harper, F. Honsel, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, 1987.

[KFFD86] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 151–161, 1986.

[Kle36] S. Kleene. λ-definability and recursiveness. *Duke Journal of Mathematics*, 2:340–353, 1936.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, CWI, The Netherlands, 1980.

[Mog92a] T. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.

[Mog92b] T. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (Charles Consel, Editor)*, pages 116–121, 1992.

[Mul89] R. Muller. *M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics.* PhD thesis, Boston University, 1989.

[Mul93] R. Muller. A staging calculus and its application to the verification of translators. Technical Report Cambridge 1, Apple Computer, 1993.

[NN88] F. Nielson and H. R. Nielson. Two-Level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[O'K92] P. O'Keefe. Private communication, 1992.

[Wan93] M. Wand. Specifying the correctness of binding-time analysis. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993. 137-143.