Staged Notational Definitions*

Walid Taha
Department of Computer Science
Rice University
taha@cs.rice.edu

Patricia Johann
Department of Computer Science
Rutgers University
pjohann@crab.rutgers.edu

ABSTRACT

Recent work proposed defining type-safe macros via interpretation into a multi-stage language. The utility of this approach is illustrated with a language called MacroML, in which all type checking is carried out before macro expansion. Building on the this work, the goal of this paper is to develop a macro language that makes it easy for programmers to reason about terms locally (i.e., without reference to context information).

This paper argues that viewing "macros as multi-stage computations" helps in developing and verifying not only type systems for macro languages but also equational reasoning principles. However, such results are not easily established with the MacroML calculus. We therefore present calculus of staged notational definitions (SND) that eliminates ad hoc features of MacroML but retains its phase distinction, and incorporates the generality of Griffin's account of notational definitions. We exhibit a formal equational theory for such a calculus and prove its soundness.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages—*Formal Definitions and Theory, Language Constructs and Features*; F.3 [**Theory of Computation**]: Logics and Meanings of Programs—*Semantics of Programming Languages, Studies of Program Constructs*

General Terms

Design, Languages, Theory, Equational Reasoning.

Keywords

Notational definitions, macros, multi-stage programming, calculi, type systems, type safety.

*Funded by NSF ITR-0113569.

1. INTRODUCTION

Macros are powerful programming constructs with applications ranging from compile-time configuration and numeric computation packages [20] to the implementation of domain-specific languages [15]. Yet the subtlety of their semantics has long been the source of an undeserved stigma:

> *"Nearly everyone agreed that macro facilities were invaluable in principle and in practice but looked down upon each particular instance as a sort of shameful family secret. If only the Right Thing could be found!"*[16].

Recently, a promising approach was proposed for the systematic formal treatment of typed generative macros as multi-stage computations [8]. Some benefits of this approach of "macros as multi-stage computations" are illustrated with MacroML, a language that supports in-lining, parametric macros, recursive macros, and macros that define new binding constructs. This approach avoids a number of technical difficulties, including any explicit reference to a gensym or fresh variable name operation. MetaOCaml is used to illustrate designing and verifying a type-safe functional language supporting macros, and for which type checking can carried out *before* macro expansion.

The goal of this paper is to demonstrate that the "macros as multi-stage computations" approach facilitates the development and verification not only of type systems but also of equational reasoning principles.

1.1 Contributions

Using ideas from Griffin's work on notational definitions in the context of logical frameworks [9], the focus of this paper is a calculus that reforms and generalizes the formal account of MacroML. Griffin's formal development of notational definitions provides almost all the machinery needed for achieving this goal. MacroML provides the phase distinction [4] that is expected from macro-definitions, in that macro expansion happens before regular computation. The main technical contribution of this paper is showing that the soundness of an equational theory for this calculus can be established directly from the equational properties of the multi-stage language used for interpretation. Compared to both the notational definitions and MacroML, the semantics of SND is defined in the untyped setting and in a compositional and context-independent manner. A key instrument in achieving this is a notion of signature which captures precisely how parameters need to be passed to macros, but which does not reintroduce additional complexity to the notion of alpha-conversion.

An alternative approach to the work presented here is to define the semantics of the macro language by interpretation into a domain-theoretic or categorical model. [1] The CPO model of Filinski [6, 7] would be too intensional with respect to second-stage computations, and so additional work would be needed to demonstrate that an equivalence holds in the second (or run-time) stage. Even when languages are extended with support for macros, the second stage is still the main stage, and the first one is often considered a preprocessing stage. Categorical models can be extensional [11, 2], but they are currently fixed to specific type systems and specific approaches to typing multi-level languages. From this point of view, interpretation into the term-model of a multi-stage language yields a result with the a new degree of generality.

1.2 Organization of this Paper

Section 3 gives a brief introduction to MacroML, and points out a technical problem that can make reasoning about programs difficult. Section 3 reviews key aspects of Griffin's development of notational definitions, and explains why this formalism alone is not sufficient for capturing the semantics of macros. Section 5 introduces the notion of a staged notational definition and presents an equational theory for a language supporting them. It is then shown that the soundness of this equational theory can be established by reflecting the MetaML equational theory through the interpretation. Section 6 presents an embedding of MacroML into SND along with a proof of its soundness. Section 7 concludes.

2. RENAMING IN MACROML

New binding constructs capture shorthands such as the following:

$$\mathsf{letopt}\ x = e_1\ \mathsf{in}\ e_2 \quad \equiv \quad \left\{ \begin{array}{l} \mathsf{case}\ e_1 \mathsf{of} \\ \quad \mathsf{Just}(x) \rightarrow e_2 \\ \quad \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing} \end{array} \right.$$

These are frequently used in both research papers (including this one) and in programs. The intent of the definition above is that whenever the pattern defined by the left hand side is encountered it should be read as an instance of the right hand side. In MacroML, the introduction of the notation defined above is achieved by the declaration

```
let mac (let opt x=e1 in e2) =
       case e1 of
          Just x  -> e2
        | Nothing -> Nothing
```

Unfortunately, with such a definition, the programmer may at some point chose to change the name of the bound variable in the right hand side of the definition from `x` to `y`, rewriting the above term to:

```
let mac (let opt x=e1 in e2) =
       case e1 of
          Just y  -> e2
        | Nothing -> Nothing
```

[1] In this paper the word "interpretation" is used to mean "translation to give meaning" rather than "an interpretor implementation".

Now the connection between the left and right hand sides of the definition is lost, and the semantics of the new term is, in fact, not defined. This renaming problem is present even in the definition of letopt above. This means that general alpha-conversion is not sound in MacroML. The absence of alpha-renaming makes it easy to introduce subtle and mistakes that can be hard to debug. At the same time, the notion of substitution depends on alpha-renaming. If alpha-renaming is non-standard, then so is substitution. Having a non-standard notion of substitution usually significantly complicates the formal reasoning principles for a calculus.

The calculus studied in this paper regains the soundness of alpha-conversion by using a *convention* associated with higher-order syntax (not higher-order syntax itself, which is already used in MacroML). This convention goes as far back as Church in the meta-theoretic level, and has been used by Griffin in a formal account of notational definitions [9], and by Michaylov and Pfenning in the context of LF [10]: whenever a term containing a free variable is used, it must be *explicitly* instantiated with the local names for these free variables. For the example above, we would write:

$$\mathsf{letopt}\ x = e_1\ \mathsf{in}\ e_2^x \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{l} \mathsf{case}\ e_1 \mathsf{of} \\ \quad \mathsf{Just}(x) \rightarrow e_2^x \\ \quad \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing} \end{array} \right.$$

With this convention, bound variables can be renamed without confusing the precise meaning of the definition. Without changing the meaning we can write the above definition as:

$$\mathsf{letopt}\ a = b\ \mathsf{in}\ c^a \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{l} \mathsf{case}\ b\ \mathsf{of} \\ \quad \mathsf{Just}(d) \rightarrow c^d \\ \quad \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing} \end{array} \right.$$

In a revised design of MacroML that avoids that renaming problem, such a definition can be written with the following concrete syntax:

```
let mac (let opt x=e1 in e2) =
       case ~e1 of
          Just x  -> ~(e2 <x>)
        | Nothing -> Nothing
```

As with MacroML, when users define new variants of existing binding constructs with established scope, there is no need to indicate on the left hand side of the `mac` declaration which variables can occur where.

Previous work on MacroML indicated a need for making explicit the escape and bracket constructs in the language, so that unfolding recursive macros can be controlled [8]. In the present work we encounter another use for these two constructs in a macro language: The escape around `e1` indicates that this is a macro-expansion time value being inserted into the template defined by the right hand side of the macro. The escape around `e2 <x>` indicates an instantiation to the free variable `x` of a macro argument `e2` containing a free variable.

3. GRIFFIN'S NOTATIONAL DEFINITIONS

In addition to the problem with renaming discussed above, in the formal account of MacroML each macro has exactly three arguments:

1. An "early" parameter, which is a regular value available at macro-expansion time

2. A "regular macro parameter", and

3. A "binder-bindee pair" representative of sub-terms of a new binding construct.

The starting point for Griffin's work abstracts away from the issue of dist-fix syntax for operators like letopt, and instead focuses on the treatment of new notation. This paper does the same, and is concerned with a calculus at the level of what Griffin calls Δ-equations. In it, new macros have a prefix-syntax, and the above example would be viewed as:

$$\mathsf{opt}\ (e_1, \lambda x.e_2) \quad \triangleq \quad \left\{ \begin{array}{l} \mathsf{case}\ \tilde{}e_1 \mathsf{of} \\ \quad \mathsf{Just}(x) \rightarrow \tilde{}(e_2\ \langle x \rangle) \\ \quad \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing} \end{array} \right.$$

Such definitions can be incorporated into a calculus in the following manner:

$$\begin{array}{rcl} p \in P_{ND} & ::= & x \mid (p,p) \mid \lambda x.p \\ e \in E_{ND} & ::= & x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i\ e \\ & \mid & \mathsf{let\text{-}delta}\ x\ p\ \stackrel{\Delta}{=} e_1\ \mathsf{in}\ e_2 \end{array}$$

Here P_{ND} is the set of *patterns* and E_{ND} is the set of *expressions*. A well-formed pattern p can have at most one occurrence of any given variable, including binding occurrences. Thus, $\mathit{Vars}(p_1) \cap \mathit{Vars}(p_2) = \emptyset$ for the pattern (p_1, p_2), and $x \notin \mathit{Vars}(p)$ for the pattern $\lambda x.p$.

Griffin's work shows how this construct can be interpreted using only the other constructs in the language. To explain this translation, Griffin defines two functions. The first function computes the binding environments of pattern terms as follows:

$$\begin{array}{lcl} \mathrm{scope}^z_z(p) & = & [] \\ \mathrm{scope}^{(p_1,p_2)}_z(p'_1, p'_2) & = & \mathrm{scope}^{p_i}_z(p'_i), \quad z \in FV(p_i) \\ \mathrm{scope}^{\lambda y.p_1}_z(\lambda x.p) & = & x :: \mathrm{scope}^{p_1}_z(p) \end{array}$$

The second function takes a pattern p and each variable z occurring free in p, and computes the sub-term of an expression which occurs in the same pattern context in which z occurs in p:

$$\begin{array}{lcl} \Phi^z_z(e) & = & e \\ \Phi^{(p_1,p_2)}_z(e) & = & \Phi^{p_i}_z(\pi_i\ e), \quad z \in FV(p_i) \\ \Phi^{\lambda y.p}_z(e) & = & \Phi^p_z(e\ y) \end{array}$$

NOTATION 1. *We write* $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ *to mean* $(\lambda x.e_2)\ e_1$.

Now Δ-equations can be interpreted as follows:

$$[\![\mathsf{let\text{-}delta}\ f\ p\ \stackrel{\Delta}{=} e_1\ \mathsf{in}\ e_2]\!] \quad = \quad \left\{ \begin{array}{l} \mathsf{let}\ y = \lambda x. \\ \quad \mathsf{let}\ v_i = \lambda \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \quad \mathsf{in}\ [\![e_1]\!] \\ \mathsf{in}\ [\![e_2]\!] \quad \boxed{\{v_i\} = FV(p)} \end{array} \right.$$

The construction $\lambda \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)$ is a nesting of lambdas with $\lambda \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)$ as the body. Note that $\lambda[].y = y$ where [] is the empty sequence.

This translation and the two supporting functions perform much of the work that is needed to provide a generic account of macros that define new binding constructs. What it does not provide is the phase distinction. Griffin's work is carried out in the context of a strongly normalizing typed lambda calculus where order of evaluation is provably irrelevant. In the context of programming languages, however, expanding macros before run-time can affect both the performance and the semantics of programs. This point is explained in detail in the context of MacroML [8]. The starting point for this paper will be a calculus that combines both the generality of Δ-equations and the phase distinction provided by a letmac construct similar to that of MacroML.

4. A MULTI-STAGE LANGUAGE

The semantics of MacroML as well as the calculus presented in this paper are defined via interpretation into a multi-stage language. This section defines a multi-stage calculus called λ-U, and presents the results on which the rest of the paper builds.

The syntax of the language is defined as follows:

$$\begin{array}{rcl} e \in E_{MetaML} & ::= & x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i\ e \\ & \mid & \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid & \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e. \end{array}$$

To develop an equational theory for this language, it is necessary to define a classification of terms that keeps track of the nesting of escaped expressions $\tilde{}e$ and bracketed expressions $\langle e \rangle$:

$$\begin{array}{rcl} e^0 \in E^0_{MetaML} & ::= & x \mid \lambda x.e^0 \mid e^0\ e^0 \mid (e^0, e^0) \mid \pi_i\ e^0 \\ & \mid & \mathsf{letrec}\ f\ x = e^0_1\ \mathsf{in}\ e^0_2 \\ & \mid & \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+1} \in E^{n+1}_{MetaML} & ::= & x \mid \lambda x.e^{n+1} \\ & \mid & e^{n+1}\ e^{n+1} \mid (e^{n+1}, e^{n+1}) \mid \pi_i\ e^{n+1} \\ & \mid & \mathsf{letrec}\ f\ x = e^{n+1}_1\ \mathsf{in}\ e^{n+1}_2 \\ & \mid & \langle e^{n+2} \rangle \mid \tilde{}e^n \mid \mathsf{run}\ e^{n+1} \\ v^0 \in V^0_{MetaML} & ::= & \lambda x.e^0 \mid (v^0, v^0) \mid \langle v^1 \rangle \\ v^{n+1} \in V^{n+1}_{MetaML} & = & E^n \end{array}$$

Values at level 0 are mostly as would be expected in a lambda calculus. Code values are not allowed to carry arbitrary terms, but rather only level 1 values. The key feature of level 1 values is that they do not contain escapes that are not surrounded by matching brackets. It turns out that in this calculus this is easy to specify: Values of level $n+1$ are exactly level n expressions. No such convenient equivalence is known for the SND calculus presented in this paper. But SND still needs a notion of level-classified expressions and values. In fact, we further need to classify expressions by a signature for the macros that can be applied to them.

Figure 1 defines the big-step semantics for λ-U. This semantics is based on similar definitions for other multi-stage languages [5, 13, 19]. There are two main features in this semantics. First, this semantics makes evaluation under lambda explicit. This, in turn, makes it easy to demonstrate that multi-stage computation often violates one of the most commonly made assumptions in programming language semantics, namely that attention can, without loss of generality, be restricted to closed terms only. Second, using just the standard notion of substitution [1], this semantics captures the *essence* of static scoping. As a result, there is no need for additional machinery to handle renaming at run-time.

The big-step semantics for MetaML is a family of partial functions $\_ \stackrel{n}{\hookrightarrow} \_ : E_{MetaML} \rightarrow E_{MetaML}$ from expressions to answers, indexed by a level n. Taking n to be 0, we can see that the first two rules correspond to the rules of a CBV lambda calculus. The rule for `run` at level 0 says that an expression is run by first evaluating it to get an expression in brackets, and then evaluating that expression. The rule for brackets at level 0 says that a bracketed expression is eval-

$$\frac{e_1 \overset{0}{\hookrightarrow} e_3 \quad e_2 \overset{0}{\hookrightarrow} e_4}{(e_1, e_2) \overset{0}{\hookrightarrow} (e_3, e_4)} \qquad \frac{e \overset{0}{\hookrightarrow} (e_1, e_2)}{\pi_i\ e \overset{0}{\hookrightarrow} e_i} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} e_3 \quad e[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_1\ e_2 \overset{0}{\hookrightarrow} e_4}$$

$$\frac{e_2[f := \lambda x.e_1[f := \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ f]] \overset{0}{\hookrightarrow} e_3}{\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} e_3} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3} \qquad \frac{}{x \overset{n+1}{\hookrightarrow} x} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{(e_1, e_2) \overset{n+1}{\hookrightarrow} (e_3, e_4)}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\pi_i\ e_1 \overset{n+1}{\hookrightarrow} \pi_i\ e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+1}{\hookrightarrow} \lambda x.e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+1}{\hookrightarrow} e_3\ e_4}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \overset{n+1}{\hookrightarrow} \mathsf{letrec}\ f\ x = e_3\ \mathsf{in}\ e_4} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+1}{\hookrightarrow} \mathsf{run}\ e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n+2}{\hookrightarrow} \tilde{}e_2} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2}$$

Figure 1: MetaML Big-Step Semantics

uated by rebuilding the expression at level 1. *Rebuilding*, or "evaluating at levels higher than 0," is intended to eliminate level 1 escapes. Rebuilding is performed by traversing the expression while correctly keeping track of levels: It simply traverses a term until a level 1 escape is encountered, at which point normal (*i.e.*, level 0) evaluation is performed. Evaluating the escaped expression must yield a bracketed expression, in which case the expression itself is returned as the value of the escaped expression.

The following results are attained using the same techniques for CBN λ-U calculus [17, 18]. These results form the basis for deriving the equational theory for the calculus of staged notational definitions.

Definition 2 (CBV λ-U Reductions). *The CBV notions of reduction of λ-U are:*

$$\begin{aligned}
\pi_i\ (v_1^0, v_2^0) \quad &\rightarrow_{\pi_U} \quad v_i^0 \\
(\lambda x.e_1^0)\ v_2^0 \quad &\rightarrow_{\beta_U} \quad e_1^0[x := v_2^0] \\
\mathit{letrec}\ f\ x = e_1^0\ \mathit{in}\ e_2^0 \quad &\rightarrow_{\mathit{rec}_U} \\
e_2^0[f := \lambda x.e_1^0[f := \mathit{letrec}\ f\ x = e_1^0\ \mathit{in}\ f]] \quad & \\
\tilde{}\langle v^1 \rangle \quad &\rightarrow_{\mathit{esc}_U} \quad v^1 \\
\mathit{run}\ \langle v^1 \rangle \quad &\rightarrow_{\mathit{run}_U} \quad v^1.
\end{aligned}$$

Definition 3 (Level 0 Termination). $\forall e \in E^0$.

$$e\Downarrow\ \equiv (\exists v \in V^0.\, e \overset{0}{\hookrightarrow} v).$$

To define the notion of observational equivalence, we need to introduce the formal notion of a context:

Definition 4 (Context). *A* context *is an expression with exactly one hole* $[]$.

$$\begin{aligned}
C \in \mathbb{C} := \quad & [] \mid (e, C) \mid (C, e) \mid \pi_i\ C \mid \\
& \lambda x.C \mid C\ e \mid e\ C \mid \\
& \mathit{letrec}\ f\ x = C\ \mathit{in}\ e \mid \\
& \mathit{letrec}\ f\ x = e\ \mathit{in}\ C \mid \\
& \langle C \rangle \mid \tilde{}C \mid \mathit{run}\ C.
\end{aligned}$$

We write $C[e]$ for the expression resulting from replacing ("filling") the hole $[]$ in the context C with the expression e.

Definition 5 (Observational Equivalence). *We define $\approx_n \in E^n \times E^n$ as follows: $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.*

$$\begin{aligned}
e_1 \approx_n e_2 \equiv \quad & \forall C \in \mathbb{C}. \\
& C[e_1], C[e_2] \in E^0 \Longrightarrow (C[e_1]\Downarrow \iff C[e_2]\Downarrow).
\end{aligned}$$

Theorem 6 (CBV λ-U Soundness). $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.

$$e_1 \rightarrow e_2 \Longrightarrow e_1 \approx_n e_2.$$

4.1 Typing MetaML

MetaML has the following types:

$$t \in T_{MetaML} \quad ::= \quad \mathsf{nat} \mid t * t \mid t \rightarrow t \mid \langle t \rangle.$$

Here nat is a type for natural numbers. Function types as usual have the form $t \rightarrow t$. The MetaML code type is denoted by $\langle t \rangle$. The soundness of this type system has already been established in [19, 13, 17]. While this type system is not the most expressive one available for MetaML (see for example [13, 17, 3]), it is simple and sufficient for our purposes.

The type system is defined by a judgment of the form $\Gamma \vdash^n e : t$. The natural number n is defined to be the *level* of the MetaML term e. The typing context Γ is a map from identifiers to types and levels, and is represented by the following term language:

$$\Gamma \quad ::= \quad [] \mid \Gamma, x : t^n.$$

In any valid context Γ there should be no repeating occurrences of the same variable name. We write $x : t^n \in \Gamma$ when $x : t^n$ is a sub-term of a valid Γ.

The rules of the type system are presented in Figure 2. The first four rules of the type system are standard typing rules, except that the level n of each term is recorded in the typing judgments. In the rules for lambda and recursive

$$\frac{x : t^n \in \Gamma}{\Gamma \vdash^n x : t} \qquad \frac{\Gamma \vdash^n e_1 : t_1 \quad \Gamma \vdash^n e_2 : t_2}{\Gamma \vdash^n (e_1, e_2) : t_1 * t_2} \qquad \frac{\Gamma \vdash^n e : t_1 * t_2}{\Gamma \vdash^n \pi_i\ e : t_i} \qquad \frac{\Gamma, x : t_1^n \vdash^n e : t_2}{\Gamma \vdash^n \lambda x.e : t_1 \to t_2} \qquad \frac{\Gamma \vdash^n e_1 : t_2 \to t \quad \Gamma \vdash^n e_2 : t_2}{\Gamma \vdash^n e_1 e_2 : t}$$

$$\frac{\begin{array}{l}\Gamma, f : (t_1 \to t_2)^n, x : t_1^n \vdash^n e_1 : t_2 \\ \Gamma, f : (t_1 \to t_2)^n \vdash^n e_2 : t_3\end{array}}{\Gamma \vdash^n \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 : t_3} \qquad \frac{\Gamma \vdash^{n+1} e : t}{\Gamma \vdash^n \langle e \rangle : \langle t \rangle} \qquad \frac{\Gamma \vdash^n e : \langle t \rangle}{\Gamma \vdash^{n+1} \tilde{e} : t} \qquad \frac{\Gamma^+ \vdash^n e : \langle t \rangle}{\Gamma \vdash^n \mathsf{run}\ e : t}$$

Figure 2: MetaML Type System

functions, the current level is taken as the level of the bound variable when it is added to the type context.

The rule for brackets gives $\langle e \rangle$ type $\langle t \rangle$ whenever e has type t and e is typed at the level one greater than the level at which $\langle e \rangle$ is typed. Thus the level of a term counts the number of surrounding brackets. The rule for escape performs the converse operation, so that escapes undo the effect of brackets. Escapes can only occur at level 1 and higher.

Finally, the rule for $\mathsf{run}\ e$ is rather subtle. We can run a term of type $\langle t \rangle$ to get a value of type t. We must, however, be careful to check that the term being run can be typed under the type context Γ^+, rather than simply in Γ. The type context Γ^+ has exactly the same variables and corresponding types as Γ, but the level of each is incremented by 1. Without this level adjustment, MetaML's type system is unsafe [19, 13, 17].

5. STAGED NOTATIONAL DEFINITIONS

We begin by presenting the syntax and semantics of SND. This extension of notational definitions has all the usual expressions for a CBV language, together with the previously described letmac construct for defining macros, pattern-bound expressions for recording macro binding information, and the explicit staging annotations $\tilde{e}$ and $\langle e \rangle$ of MacroML for controlling recursive in-lining.

5.1 Syntax and Semantics

The syntax for the SND calculus is defined as follows:

$$\begin{array}{rcl} p \in P_{SND} & ::= & x \mid \tilde{x} \mid (p, p) \mid \lambda x.p \\ q \in Q_{SND} & ::= & * \mid \tilde{*} \mid (q, q) \mid \lambda q \\ e \in E_{SND} & ::= & x \mid \lambda x.e \mid e\ e \mid (e, e) \mid \pi_i\ e \\ & \mid & \mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid & p.e \mid e_q\ e \mid \mathsf{letmac}\ f\ p\ = e_1\ \mathsf{in}\ e_2 \\ & \mid & \langle e \rangle \mid \tilde{e} \end{array}$$

An SND pattern can be either a regular macro parameter, an early macro parameter, a pair of patterns, or pair of a bound variable and a pattern. Early parameters appear as escaped variables and ensure that the arguments which replace them in a macro call are evaluated during macro expansion. Like ND patterns, an SND pattern can contain at most one occurrence of any given variable.

Elements of Q are called signatures, and capture the structure of the elements of P as defined by the following function:

$$\begin{array}{rcl} \overline{x} & = & * \\ \overline{\tilde{x}} & = & \tilde{*} \\ \overline{(p_1, p_2)} & = & (\overline{p_1}, \overline{p_2}) \\ \overline{\lambda x.p} & = & \lambda \overline{p} \end{array}$$

Signatures play an essential role in allowing us to define an untyped semantics for SND. Signatures capture precisely how parameters need to be passed to macros, but without re-introducing addition complexity to the notion of alpha-conversion.

An SND expression $p.e$ is a macro abstraction. Intuitively, it is a first-class value representing a Δ equation. This value can be used in a macro application, written $e_q\ e'$. In such a macro application, the expression e is a computation that should evaluate to a macro abstraction. Because the way parameters are passed to a macro dependens on the pattern used in the macro abstraction, the pattern p should have the signature q indicated at the application site.

5.2 Substitution

The definition of substitution for SND is standard:

$$\begin{array}{rcl} x[x := e'] & = & e' \\ y[x := e'] & = & y, \quad x \neq y \\ (e_1, e_2)[x := e'] & = & (e_1[x := e'], e_2[x := e']) \\ (\pi_i\ e)[x := e'] & = & \pi_i\ e[x := e'] \\ (\lambda y.e)[x := e'] & = & \lambda y.e[x := e'], \quad x \neq y \\ (e_1\ e_2)[x := e'] & = & e_1[x := e']\ e_2[x := e'] \\ \left(\begin{array}{l}\mathsf{letrec}\ z\ y = \\ \quad e_1 \\ \mathsf{in}\ e_2\end{array}\right)[x := e'] & = & \left(\begin{array}{l}\mathsf{letrec}\ z\ y = \\ \quad e_1[x := e'] \\ \mathsf{in}\ e_2[x := e']\end{array}\right),\ x \neq y \\ (p.e)[x := e'] & = & p.e[x := e'], \quad x \notin FV(p) \\ ((e_1)_q\ e_2)[x := e'] & = & (e_1[x := e'])_q\ e_2[x := e'] \\ \left(\begin{array}{l}\mathsf{letmac}\ f\ p = \\ \quad e_1 \\ \mathsf{in}\ e_2\end{array}\right)[x := e'] & = & \left(\begin{array}{l}\mathsf{letmac}\ f\ p = \\ \quad e_1[x := e'] \\ \mathsf{in}\ e_2[x := e']\end{array}\right),\ x \notin FV(p) \\ \langle e \rangle[x := e'] & = & \langle e[x := e'] \rangle \\ \tilde{e}[x := e'] & = & \tilde{e}[x := e'] \end{array}$$

All the side negative side conditions in this definition depend on the legitimacy of alpha-conversion. If alpha conversion cannot be performed, then substitution would be a partial function. Because of the non-standard interdependence between names of bound variables in MacroML, it is not clear how to define the notion of alpha-conversion (and in turn, substitution) in that setting.

5.3 Well-formedness

Although a type system is not necessary for defining a semantics for SND, certain well-formedness conditions are required. Intuitively, the well-formedness condition will ensure that variables intended for expansion-time use are only used in expansion-time contexts, and the same for variables intended for run-time use. The well-formedness condition will also ensure that the argument to a macro application has the appropriate form for the signature of the macro being applied.

First we define a judgment capturing variable occurrence in patterns. Let p be a pattern and x be a variable. In the context of SND we will be concerned with only two levels, the first for macro expansion, and the second for run-time.

We will let $m \in \{0,1\}$ denote these two levels. Lookup of a variable in a pattern is defined by a judgment $p \vdash^m x$ defined as follows:

$$\frac{}{x^m \vdash^m x} \quad \frac{}{(\tilde{}x)^0 \vdash^0 x} \quad \frac{p_i^0 \vdash^0 x}{(p_1,p_2)^0 \vdash^0 x} \quad \frac{p^0 \vdash^0 x}{(\lambda y.p)^0 \vdash^0 x}$$

If P is a set of patterns, write $P \vdash^m x$ to indicate that x occurs at level m in (at least) one of the patterns in P.

Let P range over sets of pairs of patterns and levels (written as p^m) where any variable occurs at most once. The *well-formedness judgments* $P \vdash^m e$ and $P \vdash^q e$ for SND expressions is defined as follows:

$$\frac{p^m \vdash^m x}{P, p^m \vdash^m x} \quad \frac{P \vdash^m e_1 \quad P \vdash^m e_2}{P \vdash^m (e_1,e_2)} \quad \frac{P \vdash^m e}{P \vdash^m \pi_i e}$$

$$\frac{P, x^m \vdash^m e}{P \vdash^m \lambda x.e} \quad \frac{P \vdash^m e_1 \quad P \vdash^m e_2}{P \vdash^m e_1\ e_2} \quad \frac{P, y^m, x^m \vdash^m e_1 \quad P, y^m \vdash^m e_2}{P \vdash^m \mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2}$$

$$\frac{P, p^0 \vdash^1 e}{P \vdash^0 p.e} \quad \frac{P \vdash^0 e_1 \quad P \vdash^q e_2}{P \vdash^1 (e_1)_q\ e_2} \quad \frac{P, p^0, f^0 \vdash^1 e_1 \quad P, f^0 \vdash^1 e_2}{P \vdash^1 \mathsf{letmac}\ f\ p\ = e_1\ \mathsf{in}\ e_2}$$

$$\frac{P \vdash^1 e}{P \vdash^0 \langle e \rangle} \quad \frac{P \vdash^0 e}{P \vdash^1 \tilde{}e}$$

$$\frac{P \vdash^1 e}{P \vdash^* e} \quad \frac{P \vdash^0 e}{P \vdash^{\tilde{}*} e} \quad \frac{P \vdash^{q_1} e_1 \quad P \vdash^{q_2} e_2}{P \vdash^{(q_1,q_2)} (e_1,e_2)} \quad \frac{P, y^1 \vdash^q e}{P \vdash^{\lambda q} \lambda y.e}$$

Well-formedness requires a context P because we need to keep track of the level at which a variable is bound to ensure it is used only at the same level. It would have been more general if well-formedness were less restrictive, as is the case, for example, with untyped MetaML which only requires expression families [17, 18]. But without this restriction only a weaker substitution property is possible, and it would not be possible to ensure that the translation is well-behaved. Well-formedness requires consistency between binding level and usage levels of variables, it requires no additional information about the types associated with each variable. A similar system was used in the context of studies on monadic multi-stage languages [12].

In the untyped setting, the signature q at the point where a macro is applied is essential for driving the well-formedness check for the argument to the macro. As will be seen later in the paper, the signature q will also drive the typing judgment for the argument in a macro application. In source programs, if we restrict e_1 in a macro application to the name of a macro, then the signature q can be inferred from the context, and does not need to be written explicitly by the programmer. At the level of the calculus, however, it is more convenient to make it explicit.

LEMMA 7 (WEAKENING). *1. If $P \vdash^n e$ then $P, p^m \vdash^n e$, and*

2. If $P \vdash^q e$ then $P, p^m \vdash^q e$.

Substitution preserves well-formedness in the following sense:

LEMMA 8. *1. If $p^n \vdash^n x$, $X, p^n \vdash^m e_1$, and $X \vdash^n e_2$, then $X, p^n \vdash^m e_1[x := e_2]$, and*

2. If $p^n \vdash^n x$, $X, p^n \vdash^q e_1$, and $X \vdash^n e_2$, then $X, p^n \vdash^q e_1[x := e_2]$.

5.4 Semantics of SND

The only change we need to make to Griffin's auxiliary functions is a straightforward extension to include early parameters:

$$\begin{array}{lcl}
\mathrm{scope}_z^z(p) & = & [] \\
\mathrm{scope}_z^{\tilde{}z}(\tilde{}p) & = & [] \\
\mathrm{scope}_z^{\lambda y.p_1}(\lambda x.p) & = & x :: \mathrm{scope}_z^{p_1}(p) \\
\mathrm{scope}_z^{(p_1,p_2)}(p_1',p_2') & = & \mathrm{scope}_z^{p_i}(p_i'), \quad z \in FV(p_i) \\
 & & \\
\Phi_z^z(e) & = & e \\
\Phi_z^{\tilde{}z}(e) & = & e \\
\Phi_z^{\lambda y.p}(e) & = & \Phi_z^p(e\ y) \\
\Phi_z^{(p_1,p_2)}(e) & = & \Phi_z^{p_i}(\pi_i\ e), \quad z \in FV(p_i)
\end{array}$$

The interpretation itself is defined as:

$$\begin{array}{rcl}
[\![x]\!]^m & = & x \\
[\![(e_1,e_2)]\!]^m & = & ([\![e_1]\!]^m, [\![e_2]\!]^m) \\
[\![\pi_i\ e]\!]^m & = & \pi_i\ [\![e]\!]^m \\
[\![\lambda x.e]\!]^m & = & \lambda x.[\![e]\!]^m \\
[\![e_1\ e_2]\!]^m & = & [\![e_1]\!]^m\ [\![e_2]\!]^m \\
[\![\mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2]\!]^m & = & \mathsf{letrec}\ y\ x = [\![e_1]\!]^m\ \mathsf{in}\ [\![e_2]\!]^m \\
[\![p.e]\!]^0 & = & \begin{cases} (\lambda x. \\ \quad \mathsf{let}\ v_i = \lambda \mathrm{scope}_{v_i}^p(p).\Phi_{v_i}^p(x) \\ \quad \mathsf{in}\ \langle [\![e]\!]^1 \rangle) & \boxed{\{v_i\} = FV(p)} \end{cases} \\
[\![(e_1)_q\ e_2]\!]^1 & = & \tilde{}([\![e_1]\!]^0\ [\![e_2]\!]^q) \\
[\![\mathsf{letmac}\ f\ p\ = e_1\ \mathsf{in}\ e_2]\!]^1 & = & \begin{cases} \tilde{}(\mathsf{letrec}\ f\ x = \\ \quad \mathsf{let}\ v_i = \lambda \mathrm{scope}_{v_i}^p(p).\Phi_{v_i}^p(x) \\ \quad \mathsf{in}\ \langle [\![e_1]\!]^1 \rangle \\ \mathsf{in}\ \langle [\![e_2]\!]^1 \rangle) & \boxed{\{v_i\} = FV(p)} \end{cases} \\
[\![\langle e \rangle]\!]^0 & = & \langle [\![e]\!]^1 \rangle \\
[\![\tilde{}e]\!]^1 & = & \tilde{}[\![e]\!]^0 \\
 & & \\
[\![e]\!]^* & = & \langle [\![e]\!]^1 \rangle \\
[\![e]\!]^{\tilde{}*} & = & [\![e]\!]^0 \\
[\![(e_1,e_2)]\!]^{(q_1,q_2)} & = & ([\![e_1]\!]^{q_1}, [\![e_2]\!]^{q_2}) \\
[\![\lambda x.e]\!]^{\lambda q} & = & \lambda x.[\![e]\!]^q[x := \tilde{}x]
\end{array}$$

5.5 Executing SND Programs

After translation, SND programs are executed in exactly the same way as MacroML programs. The result of running a well-formed SND program $\vdash^1 e$ is obtained simply by evaluating the MetaML term $\mathsf{run}\ e\langle [\![\vdash^1 e]\!] \rangle$. A finer-grained view of the evaluation of $[\![\vdash^1 e]\!]$ can be obtained by observing that evaluating proceeds in two distinct steps, namely

- Macro expansion, in which the SND program e is expanded into the MetaML program e_1' for which

$$\langle [\![\vdash^1 e]\!] \rangle \overset{0}{\hookrightarrow} e_1'.$$

- Regular execution, in which the expansion e_1' of e is evaluated to obtain the value e_2' for which

$$\mathsf{run}\ e_1' \overset{0}{\hookrightarrow} e_2'.$$

5.6 Examples

The following sequence of examples are given in the context of a hypothetical extension of our calculus, where arithmetic expressions have been added in a standard manner, such as using Church numerals.

The reader should keep in mind that what is referred to as level 0 corresponds to what is traditionally called macro-expansion time, and level 1 is what is traditionally called run time.

EXAMPLE 9 (DIRECT MACRO INVOCATION). *In SND, the term level 1 term*

$$(x.\tilde{}x + \tilde{}x)_* \; 2 + 3$$

represents a directly application of a macro $x.\tilde{}x + \tilde{}x$ to the term $2 + 3$. The macro itself simply takes on argument and constructs a term that adds this argument to itself. The result of applying the translation above to this macro can be calculated as follows:

$$\begin{array}{ll} & [\![(x.\tilde{}x + \tilde{}x)_* \; 2 + 3]\!]^1 \\ = & \tilde{}([\![(x.\tilde{}x + \tilde{}x)_*]\!]^0 \; [\![2 + 3]\!]^*) \\ = & \tilde{}(\lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle [\![(\tilde{}x + \tilde{}x)]\!]^1 \rangle \; \langle [\![2 + 3]\!]^1 \rangle) \\ = & \tilde{}(\lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle (\tilde{}x + \tilde{}x) \rangle \; \langle 2 + 3 \rangle) \end{array}$$

The result is level 1 MetaML term is which, if rebuilt at level 1, produces the term $(2 + 3) + (2 + 3)$. That is,

$$\tilde{}(\lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle (\tilde{}x + \tilde{}x) \rangle \; \langle 2 + 3 \rangle) \overset{1}{\hookrightarrow} (2 + 3) + (2 + 3)$$

EXAMPLE 10 (FIRST CLASS MACROS). *Macros can be passed around as values, and then used in the context of a macro application. The level 0 term*

$$\mathsf{let}\; M = x.\tilde{}x + \tilde{}x \; \mathsf{in}\; \langle M_* \; 2 + 3 \rangle$$

binds the variable M to the macro from the above example, and then applies M (instead of directly applying the macro) to the same term seen above. The result of translating the above SND term is as follows:

$$\begin{array}{ll} & [\![\mathsf{let}\; M = x.\tilde{}x + \tilde{}x \; \mathsf{in}\; \langle M_* \; 2 + 3 \rangle]\!]^0 \\ = & \mathsf{let}\; M = \lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle [\![(\tilde{}x + \tilde{}x)]\!]^1 \rangle \; \mathsf{in}\; \langle [\![M_* \; 2 + 3]\!]^1 \rangle \\ = & \mathsf{let}\; M = \lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle \tilde{}x + \tilde{}x \rangle \; \mathsf{in}\; \langle \tilde{}(M \; [\![2 + 3]\!]^*) \rangle \\ = & \mathsf{let}\; M = \lambda y.\mathsf{let}\; x = y \; \mathsf{in}\; \langle \tilde{}x + \tilde{}x \rangle \; \mathsf{in}\; \langle \tilde{}(M \; \langle 2 + 3 \rangle) \rangle \end{array}$$

The resulting level 0 MetaML, when evaluated at level 0, produces the term $\langle (2 + 3) + (2 + 3) \rangle$. If we had an escape $\tilde{}$ around the full SND that we translated, then it would have been a level 1 term. Translating it would produce a level 1 MetaML term, which when rebuilt at level 0, produces exactly the same result as the example above.

EXAMPLE 11 (BASIC MACRO DECLARATIONS).

$$\begin{array}{ll} & [\![\mathsf{letmac}\; M \; x = \tilde{}x + \tilde{}x \; \mathsf{in}\; M_* \; 2 + 3]\!]^1 \\ = & \tilde{}(\mathsf{letrec}\; M \; y = \mathsf{let}\; x = y \; \mathsf{in}\; \langle [\![\tilde{}x + \tilde{}x]\!]^1 \rangle \; \mathsf{in}\; \langle [\![M_* \; 2 + 3]\!] \rangle) \\ = & \tilde{}(\mathsf{letrec}\; M \; y = \mathsf{let}\; x = y \; \mathsf{in}\; \langle \tilde{}x + \tilde{}x \rangle \; \mathsf{in}\; \langle \tilde{}(M \; \langle 2 + 3 \rangle) \rangle) \end{array}$$

EXAMPLE 12 (STAGED NOTATIONAL DEFINITIONS). *We give a minimal example. Consider the following SML datatype:*

```
datatype 'a C = C of 'a
fun unC (C a) = a
```

The SND term

$$(x, \lambda y.z).(\lambda y.\tilde{}(z \; \langle y \rangle))(\mathtt{unC}\; \tilde{}x)$$

defines a "monadic-let" macro for this datatype.

$$\begin{array}{ll} & \left[\!\!\left[\begin{array}{l} \mathsf{letmac}\; L \; (x, \lambda y.z) = (\lambda y.\tilde{}(z \; \langle y \rangle))(\mathtt{unC}\; \tilde{}x) \\ \mathsf{in}\; L_{(*,\lambda *)} \; (\mathtt{C}\; 7, \lambda x.\mathtt{C}\; (x + x)) \end{array} \right]\!\!\right]^1 \\ = & \left\{ \begin{array}{l} \tilde{}(\mathsf{letrec}\; L \; x' = \\ \quad \mathsf{let}\; x = \lambda scope_x^{(x,\lambda y.z)}(x, \lambda y.z).\Phi_x^{(x,\lambda y.z)}(x') \\ \quad \mathsf{let}\; z = \lambda scope_z^{(x,\lambda y.z)}(x, \lambda y.z).\Phi_z^{(x,\lambda y.z)}(x') \\ \quad \mathsf{in}\; \langle [\![(\lambda y.\tilde{}(z \; \langle y \rangle))(\mathtt{unC}\; \tilde{}x)]\!]^1 \rangle \\ \mathsf{in}\; \langle [\![L_{(*,\lambda *)} \; (\mathtt{C}\; 7, \lambda x.\mathtt{C}\; (x + x))]\!]^1 \rangle) \end{array} \right. \\ = & \left\{ \begin{array}{l} \tilde{}(\mathsf{letrec}\; L \; x' = \\ \quad \mathsf{let}\; x = \pi_1 \; x' \\ \quad \mathsf{let}\; z = \lambda y.((\pi_2 \; x')y) \\ \quad \mathsf{in}\; \langle (\lambda y.\tilde{}(z \; \langle y \rangle))(\mathtt{unC}\; \tilde{}x) \rangle \\ \mathsf{in}\; \langle \tilde{}(L \; ([\![\mathtt{C}\; 7]\!]^*, [\![\lambda x.\mathtt{C}\; (x + x)]\!]^{\lambda *})) \rangle) \end{array} \right. \\ = & \left\{ \begin{array}{l} \tilde{}(\mathsf{letrec}\; L \; x' = \\ \quad \mathsf{let}\; x = \pi_1 \; x' \\ \quad \mathsf{let}\; z = \lambda y.((\pi_2 \; x')y) \\ \quad \mathsf{in}\; \langle (\lambda y.\tilde{}(z \; \langle y \rangle))(\mathtt{unC}\; \tilde{}x) \rangle \\ \mathsf{in}\; \langle \tilde{}(L \; (\langle \mathtt{C}\; 7 \rangle, \lambda x.\langle \mathtt{C}\; (\tilde{}x + \tilde{}x) \rangle)) \rangle) \end{array} \right. \end{array}$$

Evaluating the final term according to the standard MetaML semantics at level 1 yields

$$(\lambda y.\mathtt{C}\; (y + y)) \; (\mathtt{unC}\; (\mathtt{C}\; 7))$$

5.6.1 Basic Properties of Interpretation

LEMMA 13. *For all m, q, and e,*

1. *If $P \vdash^m e$ then $[\![e]\!]^m$ is defined and is in E^m_{MetaML}.*
2. *If $P \vdash^q e$ then $[\![e]\!]^q$ is defined and is in E^0_{MetaML}.*

The translation of SND into MetaML is substitutive, in the sense that it commutes with substitution. This is by contrast with the MetaML translation of MacroML [8].

LEMMA 14. *For all $m, n \geq 0$, for all patterns p and q, all sets P and P' of patterns, and for all expressions e_1 and e_2,*

1. *If $p^n \vdash x$, if $P, P', p^n \vdash^m e_1$, and if $P \vdash^n e_2$, then $[\![e_1]\!]^m[x := [\![e_2]\!]^n] = [\![e_1[x := e_2]]\!]^m$.*
2. *If $p^n \vdash x$, if $P, P', p^n \vdash^q e_1$, and if $P \vdash^n e_2$, then $[\![e_1]\!]^q[x := [\![e_2]\!]^n] = [\![e_1[x := e_2]]\!]^q$.*

5.7 Reasoning about SND Programs

A reasonable approach to defining equivalence on SND terms is to consider the behavior of the terms generated by the translation.

DEFINITION 15. *We write $e_1 \approx_m e_2$ when e_1 and e_2 are observationally equivalent SND terms. Two SND expressions e_1 and e_2 are* observationally equivalent *provided there exists a P such that both $P \vdash^m e_1$ and $P \vdash^m e_2$ are derivable, and $[\![e_1]\!]^m$ and $[\![e_2]\!]^m$ are level-m observationally equivalent MetaML terms.*

Specifying the equational theory requires defining five useful sets of terms. They are all subsets of the set E_{ND}:

DEFINITION 16.

$$\begin{array}{rcl} e^m \in E^m_{SND} & = & \{\; e \mid \forall e.\exists P.\; P \vdash^m e \;\} \\ e^p \in E^p_{SND} & = & \{\; e^0 \mid \forall e^0.\exists P.\; P \vdash^{\overline{p}} e^0 \;\} \\ v^0 \in V^0_{SND} & ::= & \lambda x.e^0 \mid (v^0, v^0) \mid p.e^1 \mid \langle v^1 \rangle \\ v^1 \in V^1_{SND} & ::= & x \mid \lambda x.v^1 \mid v^1 \; v^1 \mid (v^1, v^1) \mid \pi_i \; v^1 \\ & \mid & \mathsf{letrec}\; y \; x = v^1_1 \; \mathsf{in}\; v^1_2 \\ v^p \in V^p_{SND} & = & \{\; v^0 \mid \forall v^0.\exists P.\; P \vdash^{\overline{p}} v^0 \;\} \end{array}$$

The set V^1_{SND} is a subset of E_{SND}, but it does not allow for escapes, macros, or brackets in terms. The sets represent the syntax of terms after the macro expansion phase has been completed.

LEMMA 17. *The translation preserves syntactic categories in the sense that:*

1. $[\![v^m]\!]^m \in V^m_{MetaML}$
2. $[\![v^p]\!]^{\overline{p}} \in V^0_{MetaML}$

DEFINITION 18 (SND REDUCTIONS). *The relation $\rightarrow$ is defined as the reflexive transitive closure of the compatible extension of the following notions of reduction defined on SND terms:*

$$\pi_i\ (v^0_1, v^0_2) \quad \rightarrow_\pi \quad v^0_i$$

$$(\lambda x.e^0_1)v^0_2 \quad \rightarrow_\beta \quad e^0_1[x := v^0_2]$$

$$\mathit{letrec}\ f\ x = e^0_1\ \mathit{in}\ e^0_2 \rightarrow_{\mathit{rec}} e^0_2[f := \lambda x.e^0_1[f := \mathit{letrec}\ f\ x = e^0_1\ \mathit{in}\ f]]$$

$$(p.e^0_1)_{\overline{p}}\ v^p_2 \quad \rightarrow_\mu \quad e^0_1[v_i = \lambda \mathit{scope}^p_{v_i}(p).\Theta^p_{v_i}(v^p_2)],\ \{v_i\} = FV(p)$$

$$\tilde{}\langle v^1\rangle \quad \rightarrow_{\mathit{esc}} \quad v^1$$

$$\mathit{letmac}\ fp\ = e^1_1\ \mathit{in}\ e^1_2 \rightarrow_{\mathit{mac}} \begin{cases} \tilde{}(\mathit{letrec}\ f\ x = \\ \quad \mathit{let}\ v_i = \lambda \mathit{scope}^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \quad \mathit{in}\ \langle e^1_1\rangle \\ \mathit{in}\ \langle e^1_2\rangle) \quad \boxed{\{v_i\} = FV(p)} \end{cases}$$

Here,

$$\begin{array}{lcl} \Theta^z_z(e) & = & \langle e\rangle \\ \Theta^{\tilde{}z}_z(e) & = & e \\ \Theta^{\lambda y.p}_z(\lambda x.e) & = & \Theta^p_z(e[x := \tilde{}y]) \\ \Theta^{(p_1,p_2)}_z((e_1,e_2)) & = & \Theta^{p_i}_z(e_i), \quad z \in FV(p_i) \end{array}$$

The following property characterizes the essential role of the $\Theta^p_z(e)$ construction:

LEMMA 19. *1. $\Theta^p_z(e^p)$ is always defined*

2. $\Phi^p_z([\![e^p]\!]^{\overline{p}}) = [\![\Theta^p_z(e^p)]\!]^0$

In the above definition, if q were replaced by p, then the notion of reduction would be too syntactic: For alpha-conversion to be allowed, the names of the local variables occurring in p should be irrelevant to the macro application site. Avoiding this problem is the main role played by the notion of signatures.

THEOREM 20 (SOUNDNESS OF SND REDUCTIONS). *Whenever $P \vdash^m e_1$ and $P \vdash^m e_2$ then*

$$e_1 \rightarrow e_2 \Longrightarrow e_1 \approx_m e_2$$

PROOF. The main part of this proof involves establishing that the translation of the left hand side of each notion of reduction reduces (by MetaML reduction) to the translation of the right hand side. By the soundness of MetaML's reduction (and the definition of observational equivalence on SND terms) it follows that the translations of both sides are observationally equivalent. All cases will depend on Lemma 17. Standard application uses the substitution lemma. Macro application and macro definition will use Lemma 19.

Case π:

$$[\![\pi_i\ (v^0_1, v^0_2)]\!]^m = \pi_i\ ([\![v^0_1]\!]^m, [\![v^0_2]\!]^m) \rightarrow_{\pi_U} [\![v^0_i]\!]^m$$

Case β:

$$[\![(\lambda x.e^0)v^0]\!]^m = [\![(\lambda x.e^0)]\!]^m [\![v^0]\!]^m = (\lambda x.[\![e^0]\!]^m)[\![v^0]\!]^m \rightarrow_{\beta_U} [\![e^0]\!]^m[x := [\![v^0]\!]^m] = [\![e^0[x := v^0]]\!]^m$$

Case rec:

$$\begin{array}{rl} & [\![\mathsf{letrec}\ f\ x = e^0_1\ \mathsf{in}\ e^0_2]\!]^m \\ = & \mathsf{letrec}\ f\ x = [\![e^0_1]\!]^m\ \mathsf{in}\ [\![e^0_2]\!]^m \\ \rightarrow_{\mathsf{rec}_U} & [\![e^0_2]\!]^m[f := \lambda x.[\![e^0_1]\!]^m[f := \mathsf{letrec}\ f\ x = [\![e^0_1]\!]^m\ \mathsf{in}\ f]] \\ = & [\![e^0_2[f := \lambda x.e^0_1[f := \mathsf{letrec}\ f\ x = e^0_1\ \mathsf{in}\ f]]]\!]^m \end{array}$$

Case μ:

$$\begin{array}{rl} & [\![(p.e^0)_{\overline{p}}\ v^p]\!]^1 \\ = & \tilde{}((\lambda x.\mathsf{let}\ v_i = \lambda \mathsf{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle [\![e^0]\!]^1\rangle)\ [\![v^p]\!]^{\overline{p}}) \\ \rightarrow_{\beta_U} & \tilde{}(\mathsf{let}\ v_i = \lambda \mathsf{scope}^p_{v_i}(p).\Phi^p_{v_i}([\![v^p]\!]^{\overline{p}})\ \mathsf{in}\ \langle [\![e^0]\!]^1\rangle) \\ = & \tilde{}(\mathsf{let}\ v_i = \lambda \mathsf{scope}^p_{v_i}(p).[\![\Theta^p_{v_i}(v^p)]\!]^0\ \mathsf{in}\ \langle [\![e^0]\!]^1\rangle) \\ = & \tilde{}(\mathsf{let}\ v_i = [\![\lambda \mathsf{scope}^p_{v_i}(p).\Theta^p_{v_i}(v^p)]\!]^0\ \mathsf{in}\ \langle [\![e^0]\!]^1\rangle) \\ \rightarrow_{\beta_U} & \tilde{}\langle [\![e^0]\!]^1[v_i := [\![\lambda \mathsf{scope}^p_{v_i}(p).\Theta^p_{v_i}(v^p)]\!]^0]\rangle \\ = & \tilde{}\langle [\![e^0[v_i := \lambda \mathsf{scope}^p_{v_i}(p).\Theta^p_{v_i}(v^p)]]\!]^1\rangle \\ \rightarrow_{\mathsf{esc}_U} & [\![e^0[v_i := \lambda \mathsf{scope}^p_{v_i}(p).\Theta^p_{v_i}(v^p)]]\!]^1 \end{array}$$

Case esc:

$$[\![\tilde{}\langle v^1\rangle]\!]^1 = \tilde{}\langle [\![v^1]\!]^1\rangle \rightarrow_{\mathsf{esc}_U} [\![v^1]\!]^1$$

Case mac:

$$\begin{array}{rl} & [\![\mathsf{letmac}\ fp\ = e^1_1\ \mathsf{in}\ e^1_2]\!]^1 \\ = & \begin{cases} \tilde{}(\mathsf{letrec}\ f\ x = \\ \quad \mathsf{let}\ v_i = \lambda \mathsf{scope}^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \quad \mathsf{in}\ \langle [\![e^1_1]\!]^1\rangle \\ \mathsf{in}\ \langle [\![e^1_2]\!]^1\rangle) \quad \boxed{\{v_i\} = FV(p)} \end{cases} \\ = & \left[\!\!\left[\begin{array}{l} \tilde{}(\mathsf{letrec}\ f\ x = \\ \quad \mathsf{let}\ v_i = \lambda \mathsf{scope}^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \quad \mathsf{in}\ \langle e^0_1\rangle \\ \mathsf{in}\ \langle e^0_2\rangle) \quad \boxed{\{v_i\} = FV(p)} \end{array} \right]\!\!\right]^1 \end{array}$$

□

5.8 Type System for SND

The development so far shows that we can define the semantics for untyped SND, and that there is a non-trivial equation theory for this language. These equalities will hold *for any* typed variant of SND. To illustrate how such a type system is defined an verified, this section presents a simply-typed version of SND and proves that it guarantees type safety.

The set of type terms is defined by the grammar:

$$t \in T_{SND} \quad ::= \quad \mathsf{nat} \mid t \rightarrow t \mid \langle t\rangle$$

where nat is representative for various base types, $t_1 \rightarrow t_2$ is a type for partial functions that take a value of type t_1 and return a value of type t_2, and $\langle t\rangle$ is the type for a next-stage value of type t.

To define the typing judgment, a notion of type context is needed. Type contexts are generated by the following grammar, with additional the condition that any variable name occurs exactly once:

$$\Gamma \in G_{SND} \quad ::= \quad [] \mid \Gamma, x : t^m \mid \Gamma, \underline{x} : t^1 \mid \Gamma, p : t^0$$

The case of $\underline{x} : t^1$ is treated just like that of $x : t^m$ by the type system. The distinction is useful only as an instrument for proving type safety.

Figure 3 presents the types systems and various auxiliary judgments needed in defining the type system.

5.9 Type Safety for SND

Type safety for SND is established by showing that the translation of terms maps well-typed SND terms to well-typed MetaML terms (which themselves are known to be type-safe). SND types map unchanged to MetaML terms. The translation on type contexts needs to "flatten" the $p : t^0$ bindings into bindings of the form $x : t^0$. This translation also transforms bindings of the form $\underline{y} : t^0$ to ones of the form $y : \langle t \rangle$. Formally, the translation interpreters each binding in an an type context Γ as follows:

$$\begin{array}{rcl} [\![x : t^m]\!] & = & x : t^m \\ [\![\underline{x} : t^1]\!] & = & x : \langle t \rangle^0 \\ [\![\tilde{x} : t^0]\!] & = & x : t^0 \\ [\![(p_1, p_2) : (t_1, t_2)^0]\!] & = & [\![p_1 : t_1^0]\!], [\![p_2 : t_2^0]\!] \\ [\![(\lambda x.p) : (\langle t_1 \rangle \to t_2)^0]\!] & = & \{x_i : (\langle t_1 \rangle \to t_i)^0\} \\ & & \text{where } \{x_i : t_i^0\} = [\![p : t_2]\!] \end{array}$$

THEOREM 21 (TYPE SAFETY). *If $[] \vdash^m e : t$ is a valid SND judgment, then translating e to MetaML yields a well-typed MetaML program, and executing that program does not generate any MetaML run-time errors.*

PROOF. The first part is by Lemma 22, and the second part follows from the type safety property of the MetaML type system presented in previous work [19, 13]. □

LEMMA 22 (TYPE PRESERVATION). *If $[\![\Gamma]\!]$ is well-defined and y_i are the underlined variables in Γ, then*

1. *if $\Gamma \vdash^m e : t$ is a valid SND judgment, then $[\![\Gamma]\!] \vdash^m [\![e]\!]^m[y_i = \tilde{y}_i] : t$ is a valid MetaML judgment, and*
2. *if $\Gamma \vdash^q e : t$ is a valid SND judgment, then $[\![\Gamma]\!] \vdash^0 [\![e]\!]^q[y_i = \tilde{y}_i] : t$ is a valid MetaML judgment.*

Comparing this result to the corresponding result for MacroML, we notice that: 1) types require no translation, 2) the translation operates directly on the term being translated and not on the typing judgment for that term, and 3) each part of the lemma requires assuming that $[\![\Gamma]\!]$ is well-defined. This last condition is vacuously true for the empty type context (which is what is needed for type safety). In the general case, this condition simply means that any binding $p : t^m$ occurs at level $m = 0$ and satisfies the well-formedness condition $\vdash \overline{p} : t$

6. EMBEDDING OF MACROML IN SND

Embedding MacroML into SND requires a minor modification to the original definition of SND. In particular, MacroML macros are restricted to having exactly three arguments (representative of the three kinds of arguments: early parameters, regular parameters, and new binding constructs). In the original definition, these arguments are interpreted as curried. It simplifies the embedding to modify the original definition to treat these three arguments as a tuple.

The type system is unchanged, and is reproduced in Figure 5. The modified translation is presented in Figure 4.[2]

CONJECTURE 23 (EMBEDDING). *Translating any well-formed MacroML term into SND and then into MetaML is equivalent to translating the MacroML term directly into MetaML. That is, If $\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t$ is a valid MacroML judgment, then*

$$[\![[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t]\!]^M]\!]^m \approx_m [\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t]\!].$$

7. CONCLUSIONS

Previous work demonstrated that the "macros as multi-stage computations" approach is instrumental for developing and verifying type systems for an expressive macro language. The present work shows that, for a generalized revision of the MacroML calculus, the approach also facilitates developing an equational theory for a macro language. Considering this problem has also resulted in a better language, in that the semantics of programs does not change if the programmer accidentally "renames" what she perceives is a locally bound variable. The work presented here builds heavily on Griffin's work on notational definitions, and extends it to the untyped setting using the notion of signatures.

Compared to MacroML, SND embodies a number of technical improvements in terms of design of calculi for modeling macros. First, it supports alpha-equivalence. Second, its translation into MetaML is substitutive.

Compared to notational definitions, SND provides the phase distinction that is not part of the formal account of notational definitions. Introducing the phase distinction means that macro application is no longer just function application. To address this issue, a notion of signatures is introduced, and is used to define an untyped semantics and equational theory for SND.

Previous work on MacroML indicated a need for making explicit the escape and bracket constructs in the language, so that unfolding recursive macros can be controlled. In the present work, escapes and brackets are found to be useful for specifying explicitly the instantiation of a macro parameter with free variables to specific variables inside the definition of the macro. These observations, as well as the results presented in this paper, suggests that macro languages may naturally and usefully be viewed as conservative extensions of multi-stage (or at least two-level) languages.

8. REFERENCES

[1] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[2] BENAISSA, Z. E.-A., MOGGI, E., TAHA, W., AND SHEARD, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (1999).

[3] CALCAGNO, C., MOGGI, E., AND TAHA, W. Closed types as a simple approach to safe imperative

[2] At the time of submission the proof was only sketched. It would be useful to get this result, but the main result of the paper does not depend no it.

$$\frac{}{\vdash * : \langle t \rangle} \quad \frac{}{\vdash \tilde{} * : t} \quad \frac{\vdash q_1 : t_1 \quad \vdash q_2 : t_2}{\vdash (q_1, q_2) : t_1 * t_2} \quad \frac{\vdash q : t_2}{\vdash \lambda q : \langle t_1 \rangle \to t_2}$$

$$\frac{}{x : t^m \vdash^m x : t} \quad \frac{}{(\tilde{} x : t)^0 \vdash^0 x : t} \quad \frac{p_i : t_i^0 \vdash^0 x : t}{(p_1, p_2) : (t_1, t_2)^0 \vdash^0 x : t} \quad \frac{p : t_2^0 \vdash^0 x : t_3}{(\lambda y.p) : (t_1 \to t_2)^0 \vdash^0 x : t_1 \to t_3}$$

$$\frac{p : t_1^m \vdash^m x : t_2}{\Gamma, p : t_1^m, \Gamma' \vdash^m x : t_2} \quad \frac{}{\Gamma, \underline{y} : t^1, \Gamma' \vdash^1 y : t} \quad \frac{\Gamma \vdash^m e_1 : t_1 \quad \Gamma \vdash^m e_2 : t_2}{\Gamma \vdash^m (e_1, e_2) : t_1 * t_2} \quad \frac{\Gamma \vdash^m e : t_1 * t_2}{\Gamma \vdash^m \pi_i e : t_i}$$

$$\frac{\Gamma, x : t_1^m \vdash^m e : t_2}{\Gamma \vdash^m \lambda x.e : t_1 \to t_2} \quad \frac{\Gamma \vdash^m e_1 : t_1 \to t_2 \quad \Gamma \vdash^m e_2 : t_1}{\Gamma \vdash^m e_1\ e_2 : t_2} \quad \frac{\Gamma, y : t_1 \to t_2^m, x : t_1^m \vdash^m e_1 : t_2 \quad \Gamma, y : t_1 \to t_2^m \vdash^m e_2 : t_3}{\Gamma \vdash^m \mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2 : t_3}$$

$$\frac{\Gamma, p : t_1^0 \vdash^1 e : t_2 \quad \vdash \overline{p} : t_1}{\Gamma \vdash^0 p.e : t_1 \to \langle t_2 \rangle} \quad \frac{\Gamma \vdash^0 e_1 : t_1 \to \langle t_2 \rangle \quad \vdash q : t_1 \quad \Gamma \vdash^q e_2 : t_1}{\Gamma \vdash^1 (e_1)_q\ e_2 : t_2}$$

$$\frac{\Gamma, p : t_1^0, f : t_1 \to \langle t_2 \rangle^0 \vdash^1 e_1 : t_2 \quad \vdash \overline{p} : t_1 \quad \Gamma, f : t_1 \to \langle t_2 \rangle^0 \vdash^1 e_2 : t_3}{\Gamma \vdash^1 \mathsf{letmac}\ f\ p\ = e_1\ \mathsf{in}\ e_2 : t_3} \quad \frac{\Gamma \vdash^1 e : t_1}{\Gamma \vdash^0 \langle e \rangle : \langle t_1 \rangle} \quad \frac{\Gamma \vdash^0 e : \langle t_1 \rangle}{\Gamma \vdash^1 \tilde{} e : t_1}$$

$$\frac{\Gamma \vdash^1 e_1 : t_1}{\Gamma \vdash^* e_1 : \langle t_1 \rangle} \quad \frac{\Gamma \vdash^0 e : t_1}{\Gamma \vdash^{\tilde{} *} e : t_1} \quad \frac{\Gamma \vdash^{q_1} e_1 : t_1 \quad \Gamma \vdash^{q_2} e_2 : t_2}{\Gamma \vdash^{(q_1, q_2)} (e_1, e_2) : t_1 * t_2} \quad \frac{\Gamma, \underline{y} : t_1^1 \vdash^q e : t_2}{\Gamma \vdash^{\lambda q} \lambda y.e : \langle t_1 \rangle \to t_2}$$

Figure 3: Type System for SND

multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–36.

[4] Cardelli, L. Phase distinctions in type theory. (Unpublished manuscript.) Available online from http://www.luca.demon.co.uk/Bibliography.html, 1988.

[5] Davies, R. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, 1996), IEEE Computer Society Press, pp. 184–195.

[6] Filinski, A. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming (PPDP)* (1999), vol. 1702 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 378–395.

[7] Filinski, A. Normalization by evaluation for the computational lambda-calcu lus. In *Typed Lambda Calculi and Applications: 5th International Confe rence (TLCA)* (2001), vol. 2044 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 151–165.

[8] Ganz, S., Sabry, A., and Taha, W. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)* (Florence, Italy, September 2001), ACM.

[9] Griffin, T. G. Notational definitions — a formal account. In *Proceedings of the Third Symposium on Logic in Computer Science* (1988).

[10] Michaylov, S., and Pfenning, F. Natural semantics and some of its meta-theory in Elf. In *Extensions of Logic Programming* (1992), L. Hallnäs, Ed., Springer-Verlag LNCS. To appear. A preliminary version is available as Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[11] Moggi, E. Functor categories and two-level languages. In *Foundations of Software Science and Computation Structures (FoSSaCS)* (1998), vol. 1378 of *Lecture Notes in Computer Science*, Springer Verlag.

[12] Moggi, E. A monadic multi-stage metalanguage. In *Foundations of Software Science and Computation Structures (FOSSACS)* (2003), vol. 2620.

[13] Moggi, E., Taha, W., Benaissa, Z. E.-A., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207.

[14] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.

[15] Sheard, T., and Peyton-Jones, S. Template meta-programming for haskell. In *Proc. of the workshop on Haskell* (2002), ACM, pp. 1–16.

[16] Steele, Jr., G. L., and Gabriel, R. P. The evolution of LISP. In *Proceedings of the Conference on History of Programming Languages* (New York, 1993), R. L. Wexelblat, Ed., vol. 28(3) of *ACM Sigplan Notices*, ACM Press, pp. 231–270.

[17] Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [14].

[18] Taha, W. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)* (Boston, 2000), ACM Press.

[19] Taha, W., Benaissa, Z.-E.-A., and Sheard, T. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on*

Type Contexts

$$\begin{array}{rcl}
[\![\emptyset]\!] & = & \emptyset \\
[\![\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5]\!] & = & [\![\Sigma]\!], f : (t_1 * (\langle t_2 \rangle * (\langle t_3 \rangle \to \langle t_4 \rangle)) \to \langle t_5 \rangle)^0 \\
[\![\Delta, x_2 : [x_1 : t_1]t_2]\!] & = & [\![\Delta]\!], x_2 : (\langle t_1 \rangle \to \langle t_2 \rangle)^0 \\
[\![\Pi, x : t]\!] & = & [\![\Pi]\!], x : \langle t \rangle^0
\end{array}$$

Lambda Terms

$$\frac{x : t^m \in \Gamma}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m x : t]\!] = x} \qquad \frac{[\![\Sigma; \Delta; \Pi; \Gamma, x : t_1^m \vdash^m e : t_2]\!] = e'}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m \lambda x.e : t_1 \to t_2]\!] = \lambda x.e'} \qquad \frac{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 : t_2 \to t]\!] = e_1' \quad [\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e_2 : t_2]\!] = e_2'}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 e_2 : t]\!] = e_1' e_2'}$$

$$\frac{[\![\Sigma; \Delta; \Pi; \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m, x_1 : t_1^m, x_2 : t_2^m, x_3 : t_3^m \vdash^m e_1 : t]\!] = e_1' \quad [\![\Sigma; \Delta; \Pi; \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m \vdash^m e_2 : t_4]\!] = e_2'}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^m \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 : t_4]\!] = \mathsf{letrec}\ f\ x_1 = \lambda x_2.\lambda x_3.e_1'\ \mathsf{in}\ e_2'}$$

Macros

$$\frac{x : t \in \Pi}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 x : t]\!] = \tilde{}x} \qquad \frac{x_2 : [x_1 : t_1]t_2 \in \Delta \text{ and } x_1 : t_1^1 \in \Gamma}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 x_2 : t_2]\!] = \tilde{}(x_2\ \langle x_1 \rangle)}$$

$$\frac{\begin{array}{l}[\![\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta, x_2 : [x : t_3]t_4; \Pi, x_1 : t_2; \Gamma, x_0 : t_1^0 \vdash^1 e_1 : t_5]\!] = e_1' \\ [\![\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta; \Pi; \Gamma \vdash^1 e_2 : t]\!] = e_2'\end{array}}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 \mathsf{letmac}\ f(\tilde{}x_0, x_1, \lambda x.x_2)\ = e_1\ \mathsf{in}\ e_2 : t]\!] = \tilde{}(\mathsf{letrec}\ f\ x = \left(\begin{array}{l}\mathsf{let}\ x_0 = \pi_1\ x\ \mathsf{in} \\ \mathsf{let}\ x_1 = \pi_1(\pi_2\ x)\ \mathsf{in} \\ \mathsf{let}\ x_2 = \pi_2(\pi_2\ x) \\ \mathsf{in}\ \langle e_1' \rangle\end{array}\right)\ \mathsf{in}\ \langle e_2' \rangle)}$$

$$\frac{\begin{array}{l}[\![\Sigma; \Delta; \Pi; \Gamma \vdash^0 e_1 : t_1]\!] = e_1' \\ [\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 e_2 : t_2]\!] = e_2' \\ [\![\Sigma; \Delta; \Pi, x : t_3; \Gamma \vdash^1 e_3 : t_4]\!] = e_3'\end{array}}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 f(e_1, e_2, \lambda x.e_3) : t_5]\!] = \tilde{}(f\ (e_1', (\langle e_2' \rangle, \lambda x.\langle e_3' \rangle)))}\ f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5 \in \Sigma$$

Code Objects

$$\frac{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 e : t]\!] = e'}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^0 \langle e \rangle : \langle t \rangle]\!] = \langle e' \rangle} \qquad \frac{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^0 e : \langle t \rangle]\!] = e'}{[\![\Sigma; \Delta; \Pi; \Gamma \vdash^1 \tilde{}e : t]\!] = \tilde{}e'}$$

Figure 4: Modified Translation from MacroML to MetaML

Automata, Languages, and Programming (ICALP) (Aalborg, 1998), vol. 1443 of *Lecture Notes in Computer Science*, pp. 918–929.

[20] Veldhuizen, T. L. C++ templates as partial evaluation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (San Antonio, 1999), O. Danvy, Ed., University of Aarhus, Dept. of Computer Science, pp. 13–18.

APPENDIX

A. WELL-DEFINEDNESS OF TRANSLATION

The following proof is representative of the form of most of the proofs in the development presented here. An extended version of the paper will include details on various interesting aspects of the proofs underlying the development.

Proof (Lemma 13). By induction on the derivation of $P \vdash^m e$.

Part 1:

Case $\dfrac{p^m \vdash^m x}{P, p^m, P' \vdash^m x}$.

Then $[\![x]\!]^m = x$, which is always defined and is in E^m_{MetaML} since, for all variables x, $x \in E^m_{MetaML}$ for all m.

Case $\dfrac{P \vdash^m e_1 \quad P \vdash^m e_2}{P \vdash^m (e_1, e_2)}$.

Then $[\![(e_1, e_2)]\!]^m = ([\![e_1]\!]^m, [\![e_2]\!]^m)$. By the induction hypothesis, $[\![e_1]\!]^m$ is defined and in E^m_{MetaML}, and $[\![e_2]\!]^m$ is defined and in E^m_{MetaML}. Thus $([\![e_1]\!]^m, [\![e_2]\!]^m)$ is defined and in E^m_{MetaML} as well.

Case $\dfrac{P \vdash^m e}{P \vdash^m \pi_i e}$.

Then $[\![\pi_i e]\!]^m = \pi_i [\![e]\!]^m$. By the induction hypothesis, $[\![e]\!]^m$ is defined and in E^m_{MetaML}. Thus $\pi_i [\![e]\!]^m$ is defined and in E^m_{MetaML} as well.

Case $\dfrac{P, x^m \vdash^m e}{P \vdash^m \lambda x.e}$.

Then $[\![\lambda x.e]\!]^m = \lambda x.[\![e]\!]^m$. By the induction hypothesis, $[\![e]\!]^m$ is defined and in E^m_{MetaML}. Thus $\lambda x.[\![e]\!]^m$ is defined and in E^m_{MetaML} as well.

Case $\dfrac{P \vdash^m e_1 \quad P \vdash^m e_2}{P \vdash^m e_1 e_2}$.

Then $[\![e_1 e_2]\!]^m = [\![e_1]\!]^m [\![e_2]\!]^m$. By the induction hypothesis, $[\![e_1]\!]^m$ is defined and in E^m_{MetaML}, and $[\![e_2]\!]^m$ is defined and

$$\frac{x : t^m \in \Gamma}{\Sigma;\Delta;\Pi;\Gamma \vdash^m x : t} \qquad \frac{x : t \in \Pi}{\Sigma;\Delta;\Pi;\Gamma \vdash^1 x : t} \qquad \frac{x_2 : [x_1 : t_1]t_2 \in \Delta \text{ and } x_1 : t_1^1 \in \Gamma}{\Sigma;\Delta;\Pi;\Gamma \vdash^1 x_2 : t_2} \qquad \frac{\Sigma;\Delta;\Pi;\Gamma, x : t_1^m \vdash^m e : t_2}{\Sigma;\Delta;\Pi;\Gamma \vdash^m \lambda x.e : t_1 \to t_2}$$

$$\frac{\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 : t_2 \to t \quad \Sigma;\Delta;\Pi;\Gamma \vdash^m e_2 : t_2}{\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 e_2 : t} \qquad \frac{\begin{array}{l}\Gamma' \equiv \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m \\ \Sigma;\Delta;\Pi;\Gamma', x_1 : t_1^m, x_2 : t_2^m, x_3 : t_3^m \vdash^m e_1 : t \\ \Sigma;\Delta;\Pi;\Gamma' \vdash^m e_2 : t_4\end{array}}{\Sigma;\Delta;\Pi;\Gamma \vdash^m \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 : t_4} \qquad \frac{\begin{array}{l}f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5 \in \Sigma \\ \Sigma;\Delta;\Pi;\Gamma \vdash^0 e_1 : t_1 \\ \Sigma;\Delta;\Pi;\Gamma \vdash^1 e_2 : t_2 \\ \Sigma;\Delta;\Pi, x : t_3;\Gamma \vdash^1 e_3 : t_4\end{array}}{\Sigma;\Delta;\Pi;\Gamma \vdash^1 f(e_1, e_2, \lambda x.e_3) : t_5}$$

$$\frac{\begin{array}{l}\Sigma' \equiv \Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5 \\ \Sigma';\Delta, x_2 : [x : t_3]t_4;\Pi, x_1 : t_2;\Gamma, x_0 : t_1^0 \vdash^1 e_1 : t_5 \\ \Sigma';\Delta;\Pi;\Gamma \vdash^1 e_2 : t\end{array}}{\Sigma;\Delta;\Pi;\Gamma \vdash^1 \mathsf{letmac}\ f(\tilde{}x_0, x_1, \lambda x.x_2)\ = e_1\ \mathsf{in}\ e_2 : t} \qquad \frac{\Sigma;\Delta;\Pi;\Gamma \vdash^1 e : t}{\Sigma;\Delta;\Pi;\Gamma \vdash^0 \langle e\rangle : \langle t\rangle} \qquad \frac{\Sigma;\Delta;\Pi;\Gamma \vdash^0 e : \langle t\rangle}{\Sigma;\Delta;\Pi;\Gamma \vdash^1 \tilde{}e : t}$$

Figure 5: MacroML Type System

in E^m_{MetaML}. Thus $[\![e_1]\!]^m[\![e_2]\!]^m$ is defined and in E^m_{MetaML} as well.

Case $\dfrac{P, y^m, x^m \vdash^m e_1 \quad P, y^m \vdash^m e_2}{P \vdash^m \mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2}$.

Then $[\![\mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2]\!]^m = \mathsf{letrec}\ y\ x = [\![e_1]\!]^m\ \mathsf{in}\ [\![e_2]\!]^m$. By the induction hypothesis, $[\![e_1]\!]^m$ is defined and in E^m_{MetaML}, and $[\![e_2]\!]^m$ is defined and in E^m_{MetaML}. Thus $\mathsf{letrec}\ y\ x = [\![e_1]\!]^m\ \mathsf{in}\ [\![e_2]\!]^m$ is defined and in E^m_{MetaML} as well.

Case $\dfrac{P \vdash^0 e_1 \quad P \vdash^q e_2}{P \vdash^1 (e_1)_q e_2}$.

Then $[\![(e_1)_q e_2]\!]^1 = \tilde{}([\![e_1]\!]^0[\![e_2]\!]^q)$. By the induction hypothesis, $[\![e_1]\!]^0$ is defined and in E^0_{MetaML} and $[\![e_2]\!]^q$ is defined and in E^0_{MetaML} (by part 2 of the statement of the theorem). Thus $[\![e_1]\!]^0[\![e_2]\!]^q$ is defined and in E^0_{MetaML}, and so $\tilde{}([\![e_1]\!]^0[\![e_2]\!]^q)$ is defined and in E^1_{MetaML}.

Case $\dfrac{P, p^0, f^0 \vdash^1 e_1 \quad P, f^0 \vdash^1 e_2}{P \vdash^1 \mathsf{letmac}\ fp\ = e_1\ \mathsf{in}\ e_2}$.

Then $[\![\mathsf{letmac}\ fp\ = e_1\ \mathsf{in}\ e_2]\!]^1 =$
$\tilde{}(\mathsf{letrec}\ fx = \mathsf{let}\ v_i = \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle[\![e_1]\!]^1\rangle\ \mathsf{in}\ \langle[\![e_2]\!]^1\rangle)$.
By the induction hypothesis, $[\![e_1]\!]^1$ is defined and in E^1_{MetaML}, and $[\![e_2]\!]^1$ is defined and in E^1_{MetaML}. Thus $\langle[\![e_2]\!]^1\rangle \in E^0_{MetaML}$, and therefore

$$(\mathsf{letrec}\ fx = \mathsf{let}\ v_i = \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle[\![e_1]\!]^1\rangle\ \mathsf{in}\ \langle[\![e_2]\!]^1\rangle)$$

is defined and in E^0_{MetaML}. Thus $\tilde{}(\mathsf{letrec}\ fx = \mathsf{let}\ v_i = \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle[\![e_1]\!]^1\rangle\ \mathsf{in}\ \langle[\![e_2]\!]^1\rangle)$ is defined and in E^1_{MetaML}.

Case $\dfrac{P, p^0 \vdash^1 e}{P \vdash^0 p.e}$.

Then $[\![p.e]\!]^0 = \lambda x.\mathsf{let}\ v_i = \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle[\![e]\!]^1\rangle$. By the induction hypothesis, $[\![e]\!]^1$ is defined and in E^1_{MetaML}. Thus $\langle[\![e]\!]^1\rangle \in E^0_{MetaML}$, and therefore

$$\lambda x.\mathsf{let}\ v_i = \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle[\![e]\!]^1\rangle$$

is defined and in E^0_{MetaML}.

Case $\dfrac{P \vdash^1 e}{P \vdash^0 \langle e\rangle}$.

Then $[\![\langle e\rangle]\!]^0 = \langle[\![e]\!]^1\rangle$. By the induction hypothesis, $[\![e]\!]^1$ is defined and in E^1_{MetaML}, so that $\langle[\![e]\!]^1\rangle$ is defined and in E^0_{MetaML}.

Case $\dfrac{P \vdash^0 e}{P \vdash^1 \tilde{}e}$.

Then $[\![\tilde{}e]\!]^1 = \tilde{}[\![e]\!]^0$. By the induction hypothesis, $[\![e]\!]^0$ is defined and in E^0_{MetaML}, so that $\tilde{}[\![e]\!]^0$ is defined and in E^1_{MetaML}.

Part 2:

Case $\dfrac{P \vdash^1 e}{P \vdash^* e}$.

Then $[\![e]\!]^* = \langle[\![e]\!]^1\rangle$. By the induction hypothesis, $[\![e]\!]^1$ is defined and in E^1_{MetaML}. Thus $\langle[\![e]\!]^1\rangle$ is defined and in E^0_{MetaML}.

Case $\dfrac{P \vdash^0 e}{P \vdash^{\tilde{}*} e}$.

Then $[\![e]\!]^{\tilde{}*} = [\![e]\!]^0$. By the induction hypothesis, $[\![e]\!]^0$ is defined and in E^0_{MetaML}.

Case $\dfrac{P \vdash^{q_1} e_1 \quad P \vdash^{q_2} e_2}{P \vdash^{(q_1,q_2)} (e_1, e_2)}$.

Then $[\![(e_1, e_2)]\!]^{(q_1,q_2)} = ([\![e_1]\!]^{q_1}, [\![e_2]\!]^{q_2})$. By the induction hypothesis, we have that $[\![e_1]\!]^{q_1}$ is defined and in E^0_{MetaML}, and $[\![e_2]\!]^{q_2}$ is defined and in E^0_{MetaML}. Thus $([\![e_1]\!]^{q_1}, [\![e_2]\!]^{q_2})$ is defined and in E^0_{MetaML}.

Case $\dfrac{P, y^1 \vdash^q e}{P \vdash^{\lambda q} \lambda y.e}$.

Then $[\![\lambda y.e]\!]^{\lambda q} = \lambda y.[\![e]\!]^q[y := \tilde{}y]$. By the induction hypothesis, we have that $[\![e]\!]^q$ is defined and in E^0_{MetaML}. Since every MetaML variable is in E^m_{MetaML} for every m, we know that $y \in E^0_{MetaML} \cap E^1_{MetaML}$. Thus, $\tilde{}y \in E^1_{MetaML}$, and therefore $[\![e]\!]^q[y := \tilde{}y]$ is defined and in E^0_{MetaML}. But the translation maps every free variable used at level 1 to a free variable used at level 1. In the original term e, y is used at level 1. Thus, the substitution replaces a level 1 term by a level 1 term, and does not affect the level of the result of the translation. In other words, $\lambda y.[\![e]\!]^q[y := \tilde{}y] \in E^0_{MetaML}$. □

Lambda Terms

$$\frac{x : t^m \in \Gamma}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m x : t]\!]^M = x} \qquad \frac{[\![\Sigma;\Delta;\Pi;\Gamma, x : t_1^m \vdash^m e : t_2]\!]^M = e'}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m \lambda x.e : t_1 \to t_2]\!]^M = \lambda x.e'} \qquad \frac{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 : t_2 \to t]\!]^M = e_1' \quad [\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_2 : t_2]\!]^M = e_2'}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 e_2 : t]\!]^M = e_1' e_2'}$$

$$\frac{\begin{array}{l}[\![\Sigma;\Delta;\Pi;\Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m, x_1 : t_1^m, x_2 : t_2^m, x_3 : t_3^m \vdash^m e_1 : t]\!]^M = e_1' \\ [\![\Sigma;\Delta;\Pi;\Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m \vdash^m e_2 : t_4]\!]^M = e_2'\end{array}}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 : t_4]\!]^M = \mathsf{letrec}\ f\ x = e_1'\ \mathsf{in}\ e_2'}$$

Macros

$$\frac{x : t \in \Pi}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 x : t]\!]^M = \tilde{}x} \qquad \frac{x_2 : [x_1 : t_1]t_2 \in \Delta \text{ and } x_1 : t_1^1 \in \Gamma}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 x_2 : t_2]\!]^M = \tilde{}(x_2\ \langle x_1 \rangle)}$$

$$\frac{\begin{array}{l}[\![\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta, x_2 : [x : t_3]t_4; \Pi, x_1 : t_2; \Gamma, x_0 : t_1^0 \vdash^1 e_1 : t_5]\!]^M = e_1' \\ [\![\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta; \Pi; \Gamma \vdash^1 e_2 : t]\!]^M = e_2'\end{array}}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 \mathsf{letmac}\ f(\tilde{}x_0, x_1, \lambda x.x_2)\ = e_1\ \mathsf{in}\ e_2 : t]\!]^M = \mathsf{letmac}\ f(\tilde{}x_0, (x_1, \lambda x.x_2))\ = e_1\ \mathsf{in}\ e_2}$$

$$\frac{\begin{array}{l}[\![\Sigma;\Delta;\Pi;\Gamma \vdash^0 e_1 : t_1]\!]^M = e_1' \\ [\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 e_2 : t_2]\!]^M = e_2' \\ [\![\Sigma;\Delta;\Pi, x : t_3;\Gamma \vdash^1 e_3 : t_4]\!]^M = e_3'\end{array}}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 f(e_1, e_2, \lambda x.e_3) : t_5]\!]^M = (f_{(\tilde{}*,(*,\lambda*))}\ (e_1', (e_2', \lambda x.e_3'))} \quad f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5 \in \Sigma$$

Code Objects

$$\frac{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 e : t]\!]^M = e'}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^0 \langle e \rangle : \langle t \rangle]\!]^M = \langle e' \rangle} \qquad \frac{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^0 e : \langle t \rangle]\!]^M = e'}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^1 \tilde{}e : t]\!]^M = \tilde{}e'}$$

Figure 6: Translating MacroML to SND