Staged Notational Definitions

By: Walid Taha and **Patricia Johann**

Abstract

Recent work proposed defining type-safe macros via interpretation into a multi-stage language. The utility of this approach was illustrated with a language called MacroML, in which all type checking is carried out before macro expansion. Building on this work, the goal of this paper is to develop a macro language that makes it easy for programmers to reason about terms locally. We show that defining the semantics of macros in this manner helps in developing and verifying not only type systems for macro languages but also equational reasoning principles. Because the MacroML calculus is sensitive to renaming of (what appear locally to be) bound variables, we present a calculus of staged notational definitions (SND) that eliminates the renaming problem but retains MacroML's phase distinction. Additionally, SND incorporates the generality of Griffin's account of notational definitions. We exhibit a formal equational theory for SND and prove its soundness.

Taha, W. and **Johann, P.** (2003). Staged Notational Definitions. Proceedings, Generative Programming and Component Engineering 2003 (GPCE '03), pp. 97-116. NC Docks permission to re-print granted by author(s).

Staged Notational Definitions

Walid Taha*[1] and Patricia Johann[2]

[1] Department of Computer Science, Rice University, taha@cs.rice.edu
[2] 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 was illustrated with a language called MacroML, in which all type checking is carried out before macro expansion. Building on this work, the goal of this paper is to develop a macro language that makes it easy for programmers to reason about terms locally. We show that defining the semantics of macros in this manner helps in developing and verifying not only type systems for macro languages but also equational reasoning principles. Because the MacroML calculus is sensetive to renaming of (what appear locally to be) bound variables, we present a calculus of staged notational definitions (SND) that eliminates the renaming problem but retains MacroML's phase distinction. Additionally, SND incorporates the generality of Griffin's account of notational definitions. We exhibit a formal equational theory for SND and prove its soundness.

1 Introduction

Macros are powerful programming constructs with applications ranging from compile-time configuration and numeric computation packages to the implementation of domain-specific languages [14]. Yet the subtlety of their semantics has long been the source of undeserved stigma. As remarked by Steele and Gabriel,

> *"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!"* [15].

Recently, a promising approach to the semantics of generative macros has been proposed. This approach is based on the systematic formal treatment of macros as multi-stage computations [7]. Some of its benefits were illustrated with MacroML, an extension of ML which supports inlining, parametric macros, recursive macros, and macros that define new binding constructs, and in which all type-checking is carried out before macro expansion.

Interpreting macros as multi-stage computations avoids some technical difficulties usually associated with macro systems (such as hygiene and scoping issues). But while MacroML is capable of *expressing* staged macros, it is not well-suited to *reasoning* about them. This is in part because MacroML does not allow general alpha-conversion, and thus requires a non-standard notion of substitution as well.

* Funded by NSF ITR-0113569.

Our goal in this paper is to demonstrate that interpreting macros as multi-stage computations facilitates both the development of type systems *and* the verification of formal reasoning principles for macro languages. More specifically, we aim to formalize staged macros in a way that makes it possible for programmers to reason effectively about programs which use them.

1.1 Contributions

To this end we define SND, a calculus of staged notational definitions that reforms and generalizes the formal account of MacroML. SND combines the staging of MacroML with a staged adaptation of Griffin's formal development of notational definitions in the context of logical frameworks [8]. The former provides the phase distinction [3] expected from macro definitions, and ensures that SND macro expansion takes place before regular computation. The latter captures precisely how parameters need to be passed to macros. A novel notion of signature allows us to express in the term language the information needed to pass macro parameters to macros correctly. The signatures are expressive enough to allow defining macros that introduce new binding constructs.

The semantics of SND is defined by interpreting it into a multi-stage language. By contrast with both Griffin's notational definitions and MacroML, the interpretation and formal equational theory for SND are developed in an untyped setting. The interpretation of SND is defined in a compositional and context-independent manner. This makes possible the main technical contribution of this paper, namely showing that the soundness of SND's equational theory can be established directly from the equational properties of the multi-stage language used for interpretation. Our soundness result is obtained for untyped SND, and so necessarily holds for *any* typed variant of that calculus. We develop a simply typed variant of SND, prove that it is type-safe, and exhibit a sound embedding of MacroML in it.

An alternative to the approach to "macros as multi-stage computations" presented here is to define the semantics of the macro language by interpretation into a domain-theoretic or categorical model.[3] But the CPO model of Filinski [5, 6] would be too intensional with respect to second stage (i.e., run-time) computations, and so additional work would still be needed to demonstrate that equivalence holds in the second stage. This is significant: Even when languages are extended with support for macros, the second stage is still considered the primary stage of computation, with the first stage often regarded as a pre-processing stage. Categorical models can, of course, be extensional [10, 2], but they are currently fixed to specific type systems and specific approaches to typing multi-level languages. By contrast, interpreting a macro language in the term-model of a multi-stage language constitutes what we expect to be a widely applicable approach.

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

1.2 Organization of this Paper

Section 2 reviews MacroML, and illustrates how the absence of alpha-conversion 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 4 reviews λ^U, the multi-stage calclus which will serve as the interpretation language for SND. Section 5 introduces the notion of a staged notational definition and the macro language SND which supports such definitions. It also defines the interpretation of SND into λ^U, and uses this to show how SND programs are executed. Section 6 presents a formal equational theory for SND and shows that the soundness of this theory be established by reflecting the equational theory for the interpretation language through the interpretation. Section 7 presents an embedding of MacroML into SND along with a soundness result for the embedding. Section 8 concludes.

2 Renaming in MacroML

New binding constructs can be used to capture shorthands such as the following:

$$\mathsf{letopt}\ x = e_1\ \mathsf{in}\ e_2 \quad \overset{\text{def}}{=} \quad \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Just}(x) \rightarrow e_2 \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing}$$

Such notational definitions frequently appear in research papers (including this one). The intent of the definition above is that whenever the pattern defined by its left-hand side is encountered, this pattern should be read as an instance of the right-hand side.

In MacroML, the notation defined above is introduced using the declaration

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

Unfortunately, the programmer may at some point choose to change the name of the bound variable in the right-hand side of the above definition from `x` to `y`, thus rewriting the above term to

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

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, and it shows that general alpha-conversion is not sound in MacroML.

The absence of alpha-conversion makes it easy to introduce subtle mistakes that can be hard to debug. In addition, since the notion of substitution depends on alpha-conversion, MacroML necessarily has a non-standard notion of substitution. Having a non-standard notion of substitution can significantly complicate the formal reasoning principles for a calculus.

The SND calculus presented here avoids the difficulties associated with alpha-conversion. It regains the soundness of alpha-conversion by using a *convention* associated with higher-order syntax. This convention is distinct from higher-order syntax itself, which is already used in MacroML and SND. It goes as far back as Church at the meta-theoretic level, and is used in Griffin's formal account of notational definitions [8], and by Michaylov and Pfenning in the context of LF [9]. It requires that whenever a term containing free variables is used, that term must be *explicitly* instantiated with the local names for those free variables.

According to the convention, we would write the example above as

$$\mathsf{letopt}\ x = e_1\ \mathsf{in}\ e_2^x \quad \overset{\text{def}}{=} \quad \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Just}(x) \rightarrow e_2^x\ \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing}$$

It is now possible to rename bound variables without confusing the precise meaning of the definition. Indeed, the meaning of the definition is preserved if we use alpha-conversion to rewrite it to

$$\mathsf{letopt}\ x = e_1\ \mathsf{in}\ e_2^x \quad \overset{\text{def}}{=} \quad \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Just}(z) \rightarrow e_2^z\ \mid \mathsf{Nothing} \rightarrow \mathsf{Nothing}$$

In SND (extended with a `case` construct and datatypes) the above definition can be written with the following concrete syntax:

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

By contrast with MacroML, when a user defines a new variant of an existing binding construct with established scope in SND, it is necessary to indicate on the left-hand side of the `mac` declaration which variables can occur where.

Previous work on MacroML has established that explicit escape and bracket constructs can be used to control the unfolding of recursive macros [7]. In this paper we encounter additional uses for these constructs in a macro language. The escape around `e1` indicates that `e1` is a macro-expansion-time value being inserted into the template defined by the right-hand side of the macro. The escapes around the occurrences of `e2 <x>`, on the other hand, indicate an instantiation to the free variable `x` of a macro argument `e2` containing a free variable.

3 Griffin's Notational Definitions

Like Griffin's work, this paper is concerned with the formal treatment of new notation. Griffin uses a notion of term pattern to explicitly specify the binding structure of, and name the components of, the notation being defined. His formal account of notational definitions provides the technical machinery needed to manage variable bindings in patterns, including binding patterns that themselves contain bindings. A staged version is used in Section 5 to indicate precisely how parameters should be passed to macros in order to ensure that the standard notion of alpha-conversion remains valid for SND.

SND is concerned with definitions akin to what Griffin calls Δ-equations.

The above example would be written in the form of a Δ-equation as:

$$\mathsf{opt}\ (e_1, \lambda x.e_2) \quad \overset{\Delta}{=} \quad \mathsf{case}\ \tilde{}e_1\ \mathsf{of}\ \mathsf{Just}(x) \to \tilde{}(e_2\ \langle x \rangle)\ \mid \mathsf{Nothing} \to \mathsf{Nothing}$$

Δ-equations can be incorporated into a lambda calculus as follows: [4]

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

Here P_{ND} is the set of *patterns* and E_{ND} is the set of *expressions*. The pattern $\lambda x.p$ denotes a pattern in which x is bound. A well-formed pattern 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 $\lambda x.p$.

Griffin shows how the let-delta construct can be interpreted using only the other constructs in the language. To define the translation, he uses two auxiliary functions. The first computes the binding environments of pattern terms.

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

The second computes, for a pattern p, each variable z occurring free in p, and each expression e, the subterm of e which occurs in the same pattern context in which z occurs in p.

$$\Phi_z^z(e) = e, \quad \Phi_z^{(p_1,p_2)}(e) = \Phi_z^{p_i}(\pi_i\ e) \text{ where } z \in FV(p_i), \quad \Phi_z^{\lambda y.p}(e) = \Phi_z^p(e\ y)$$

We write $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ to mean $(\lambda x.e_2)\ e_1$, and $\mathsf{let}\ x_i = e_i\ \mathsf{in}\ e$ to denote a sequence of multiple simultaneous let bindings. With this notation, Δ-equations can be interpreted in terms of the other constructs of the above calculus by

$$[\![\mathsf{let\text{-}delta}\ f\ p\ \overset{\Delta}{=} e_1\ \mathsf{in}\ e_2]\!] = \begin{cases} \mathsf{let}\ f = \lambda x.\ \mathsf{let}\ v_i = \lambda \mathrm{scope}_{v_i}^p(p).\Phi_{v_i}^p(x) \\ \qquad\qquad \mathsf{in}\ [\![e_1]\!] \\ \mathsf{in}\ [\![e_2]\!] \text{ where } \{v_i\} = FV(p) \end{cases}$$

The construction $\lambda \mathrm{scope}_{v_i}^p(p).\Phi_{v_i}^p(x)$ is a nesting of lambdas with $\Phi_{v_i}^p(x)$ as the body. In $\lambda \mathrm{scope}_{v_i}^p(p).\Phi_{v_i}^p(x)$ it is critical that all three occurrences of p denote the same pattern. Note that $\lambda[\,].y = y$ where $[\,]$ is the empty sequence.

Griffin's translation performs 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 discussed in detail in the context of MacroML in [7].

[4] Throughout this paper we adopt Barendregt's convention of assuming that the set of bound variables in a given formula is distinct from the set of free variables.

4 A Multi-stage Language

The semantics of SND is given by interpretation into λ^U, a multi-stage calculus. This section presents λ^U together with the results on which the rest of the paper builds.

The syntax of λ^U is defined as follows:

$$e \in E_{\lambda^U} ::= x \mid \lambda x.e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid \mathsf{letrec}\ f\ x = e\ \mathsf{in}\ e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e$$

To develop an equational theory for λ^U, it is necessary to define a level classification of terms that keeps track of the nesting of escaped expressions $\tilde{}e$ and bracketed expressions $\langle e \rangle$. We have

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

Above and throughout this paper, the level of a term is indicated by a superscript, e and its subscripted versions to denote arbitrary λ^U expressions, and v and its subscripted versions indicate λ^U values. 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 this is easy to specify for λ^U, since values of level $n + 1$ are exactly level n expressions.

Figure 1 defines the big-step semantics for λ^U. This semantics is based on similar definitions for other multi-stage languages [4, 12, 18]. There are two features of this semantics worthy of special attention. First, it makes evaluation under lambda explicit. This shows 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. 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 λ^U is a family of partial functions $\_ \overset{n}{\hookrightarrow} \_ : E_{\lambda^U} \to E_{\lambda^U}$ from expressions to answers, indexed by level. Focusing on reductions at level 0, we see that the third and fourth 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. As a special case of the rule for evaluating bracketed expressions, we see that an expression $\langle e_1 \rangle$ is evaluated at level 0 by rebuilding e_1 at level 1. The semantics of pairing and recursive unfolding at level 0 is standard.

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

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} e_3 \quad e[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_1\ e_2 \overset{0}{\hookrightarrow} e_4}$$

$$\frac{e_2[f := \lambda x.e_1[f := \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ f]] \overset{0}{\hookrightarrow} e_3}{\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} e_3} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3}$$

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

$$\frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+1}{\hookrightarrow} e_3\ e_4} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{\begin{array}{l}\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \\ \overset{n+1}{\hookrightarrow} \mathsf{letrec}\ f\ x = e_3\ \mathsf{in}\ e_4\end{array}} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+1}{\hookrightarrow} \mathsf{run}\ e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n+2}{\hookrightarrow} \tilde{}e_2} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2}$$

Fig. 1. λ^U Big-Step Semantics

Rebuilding, i.e., 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, without performing any reductions, until a level 1 escape is encountered. When an escaped expression $\tilde{}e_1$ is encountered at level 1, normal (*i.e.*, level 0) evaluation is performed on e_1. In this case, evaluating e_1 must yield a bracketed expression $\langle e_2 \rangle$, and then e_2 is returned as the value of $\tilde{}e_1$.

In this paper we prove the soundness of the equational theory for SND by building on the following results for λ^U [17] leading upto Theorem 1:

Definition 1 (λ^U Reductions). *The notions of reduction of λ^U are:*

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

We write $\rightarrow_{\lambda^U}$ for the compatible extension of the union of these rules.

Definition 2 (Level 0 Termination). $\forall e \in E^0.\ e\Downarrow\ \equiv\ (\exists v \in V^0.\ e \overset{0}{\hookrightarrow} v)$

Definition 3 (Context). *A* context *is an expression with exactly one hole* $[\,]$.

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

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

$$\frac{\Gamma \vdash^n e_1 : t_2 \to t \quad \Gamma \vdash^n e_2 : t_2}{\Gamma \vdash^n e_1 e_2 : t}$$

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

Fig. 2. λ^U Type System

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

Definition 4 (Observational Equivalence). *The relation* $\approx_n \subseteq E^n \times E^n$ *is defined by:* $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.

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

Theorem 1 (Soundness). $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n. e_1 \to_{\lambda^U} e_2 \Longrightarrow e_1 \approx_n e_2$.

A simple type system can be defined for λ^U using the following types:

$$t \in T_{\lambda^U} ::= \mathsf{nat} \mid t * t \mid t \to t \mid \langle t \rangle$$

Here nat is a type for natural numbers, pair types have the form t_1 * t_2 and function types have the form $t_1 \to t_2$. The λ^U code type is denoted by $\langle t \rangle$.

The rules of the type system are presented in Figure 2. The type system for λ^U 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 λ^U term e. The typing context Γ is a map from identifiers to types and levels, and is represented by the term language $\Gamma ::= [] \mid \Gamma, x : t^n$. In any valid context Γ there should be no repeating occurrences of the same variable name. We write $x : t^n \in \Gamma$ if $x : t^n$ is a subterm of a valid Γ.

The first six 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 abstractions and recursive functions, the current level is taken as the level of the bound variable when it is added to the typing 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. The rule for escape performs the converse operation, so that escapes undo the effect of brackets. The level of a term thus counts the number of surrounding brackets minus the number of surrounding escapes. Escapes can only occur at level 1 and higher.

The rule for 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 an appropriate extension Γ^+ of the current type context

Γ, rather than simply in Γ itself. 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, the type system is unsafe [18, 12, 16].

The soundness of this type system has already been established [18, 12, 16]. While this type system is not the most expressive one available for λ^U (c.f. [19]), it is simple and sufficient for our purposes.

5 Staged Notational Definitions

We begin with the syntax of SND. This extension of Griffin's 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 λ^U for controlling recursive inlining. SND is defined by:

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

Elements of P_{SND} are called *patterns*. An SND pattern can be either a regular macro parameter x, an early macro parameter $\tilde{\ }x$, a pair (p_1, p_2) of patterns, or a pair $\lambda x.p$ of a bound variable and a pattern. Early parameters represent regular values which are available at macro-expansion time; they appear as escaped variables, which ensures that the arguments replacing them in a macro call are evaluated during macro expansion. Binder-bindee pairs represent subterms of new binding constructs. Like patterns in Griffin's notational definitions, an SND pattern can contain at most one occurrence of any given variable.

Elements of Q_{SND} are called *signatures*. Signatures capture the structure of the elements of P_{SND} as defined by the following function:

$$\overline{x} = * \qquad \overline{\tilde{\ }x} = \tilde{\ }* \qquad \overline{(p_1,p_2)} = (\overline{p_1},\overline{p_2}) \qquad \overline{\lambda x.p} = \lambda\overline{p}$$

Signatures play an essential role in defining an untyped semantics for SND. They capture precisely how parameters need to be passed to macros, and do this without introducing additional complexity to the notion of alpha-conversion.

Elements of E_{SND} are SND *expressions*. Of particular importance are SND macro abstractions and applications, i.e., expressions of the form $p.e$ and $e'_q\ e''$, respectively. Intuitively, a macro expression is a first-class value representing a Δ-equation. Such a value can be used in any macro application. In a macro application $e'_q\ e''$, the expression e'_q is a computation that should evaluate to a macro abstraction $p.e$. Because the way in which parameters are passed to a macro depends on the pattern used in the macro abstraction, the pattern p in $p.e$ must have the signature q indicated at the application site in $e'_q\ e''$.

The definition of substitution for SND is standard (omitted for space).

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. The same difficulties do not arise for SND because alpha-conversion is valid for SND.

Although a type system is not necessary for defining the semantics of SND, certain well-formedness conditions are required. Intuitively, the well-formedness conditions ensure that variables intended for expansion-time use are only used in expansion-time contexts, and similarly for variables intended for run-time use. The well-formedness conditions also ensure that an argument to a macro application has the appropriate form for the signature of the macro being applied. A system similar to that below has been used in the context of studies on monadic multi-stage languages [11].

Well-formedness of SND expressions requires consistency between binding levels and usage levels of variables. Thus, to define the well-formedness judgment for SND, we first need to define a judgment capturing variable occurrence in patterns. In the remainder of this paper, $m \in \{0, 1\}$ will range over evaluation levels 0 and 1. Level 0 corresponds to what is traditionally called macro-expansion time, and level 1 corresponds to what is traditionally called run-time.

Lookup of a variable x in a level-annotated pattern p is given by the judgment $p \vdash^m x$ defined as follows:

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

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

We now define the *well-formedness judgments* $P \vdash^m e$ and $P \vdash^q e$ for SND expressions. Here, the context P ranges over sets of level-annotated patterns in which any variable occurs at most once. Well-formedness must be defined with respect to a context P because we need to keep track of the level at which a variable is bound to ensure that it is used only at the same level.

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

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

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

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

As is customary, we write $\vdash^m e$ for $\emptyset \vdash^m e$.

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 also drives the typing judgment for the argument in a macro application. In source programs, if we restrict e_q in

$$
\begin{gathered}
[\![x]\!]^m = x, \quad [\![(e_1, e_2)]\!]^m = ([\![e_1]\!]^m, [\![e_2]\!]^m), \quad [\![\pi_i\ e]\!]^m = \pi_i\ [\![e]\!]^m, \quad [\![\lambda x.e]\!]^m = \lambda x.[\![e]\!]^m, \\
[\![e_1\ e_2]\!]^m = [\![e_1]\!]^m\ [\![e_2]\!]^m, \quad [\![\mathsf{letrec}\ y\ x = e_1\ \mathsf{in}\ e_2]\!]^m = \mathsf{letrec}\ y\ x = [\![e_1]\!]^m\ \mathsf{in}\ [\![e_2]\!]^m, \\
[\![p.e]\!]^0 = \lambda x.\mathsf{let}\ v_i = \lambda \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x)\ \mathsf{in}\ \langle [\![e]\!]^1 \rangle \text{ where } \{v_i\} = FV(p), \\
[\![(e_1)_q\ e_2]\!]^1 = \tilde{}([\![e_1]\!]^0\ [\![e_2]\!]^q) \\
[\![\mathsf{letmac}\ f\ p\ = e_1\ \mathsf{in}\ e_2]\!]^1 = \begin{cases} \tilde{}(\mathsf{letrec}\ f\ x = \ \mathsf{let}\ v_i = \lambda \mathrm{scope}^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \qquad\qquad\qquad\quad \mathsf{in}\ \langle [\![e_1]\!]^1 \rangle \\ \mathsf{in}\ \langle [\![e_2]\!]^1 \rangle) \text{ where } \{v_i\} = FV(p) \end{cases} \\
[\![\langle e \rangle]\!]^0 = \langle [\![e]\!]^1 \rangle, \quad [\![\tilde{}e]\!]^1 = \tilde{}[\![e]\!]^0
\end{gathered}
$$

$$
\begin{gathered}
[\![e]\!]^* = \langle [\![e]\!]^1 \rangle, \quad [\![e]\!]^{\tilde{}*} = [\![e]\!]^0, \quad [\![(e_1, e_2)]\!]^{(q_1, q_2)} = ([\![e_1]\!]^{q_1}, [\![e_2]\!]^{q_2}), \\
[\![\lambda x.e]\!]^{\lambda q} = \lambda x.[\![e]\!]^q[x := \tilde{}x]
\end{gathered}
$$

Fig. 3. Interpretation of SND in λ^U

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.

The expected weakening results hold, namely, if $P \vdash^n e$ then $P, p^m \vdash^n e$, and if $P \vdash^q e$ then $P, p^m \vdash^q e$. Substitution is also well-behaved:

Lemma 1 (Substitution Lemma).

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

5.1 Semantics of SND

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

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

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

The interpretation of SND in λ^U is given in Figure 3. It is well-behaved in the sense that, for all m, and for all signatures q and expressions e, if $P \vdash^m e$ then $[\![e]\!]^m$ is defined and is in $E^m_{\lambda^U}$, and if $P \vdash^q e$ then $[\![e]\!]^q$ is defined and is in $E^0_{\lambda^U}$. It is also substitutive:

Lemma 2. *For all m and n, and for all patterns p, signatures q, sets P and P' of patterns, and 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 $[\![e_1]\!]^q[x := [\![e_2]\!]^n] = [\![e_1[x := e_2]]\!]^q$.*

Note that no analogue of Lemma 2 holds for the interpretation of MacroML.

5.2 Executing SND Programs

After translation into λ^U, 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 λ^U 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

1. macro expansion, in which the SND program e is expanded into the λ^U program e_1' for which $\langle[\![\vdash^1 e]\!]\rangle \overset{0}{\hookrightarrow} e_1'$.
2. regular execution, in which the λ^U expansion e_1' of e is evaluated to obtain the value e_2' for which $\mathsf{run}\ e_1' \overset{0}{\hookrightarrow} e_2'$.

The following examples illustrate SND program execution. They assume a hypothetical extension of our calculus in which arithmetic expressions have been added in a standard manner, such as using Church numerals.

Example 1 (Direct Macro Invocation). In SND, the level 1 term $(x.\tilde{}x+\tilde{}x)_*\ (2+3)$ represents a direct application of a macro $x.\tilde{}x+\tilde{}x$ to the term $2+3$. The macro itself simply takes one argument and constructs a term that adds this argument to itself. The result of applying the translation to this macro is

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

The result is a level 1 λ^U term 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 2 (First Class Macros). This example demonstrates that macros can be passed around as values, and then used in the context of a macro application. The level 0 SND 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 in Example 1. The result of translating this SND term is

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

When the resulting level 0 λ^U term is evaluated at level 0, the term $\langle(2{+}3){+}(2{+}3)\rangle$ is produced. If we had put an escape ˜ around the original SND expression, then it would have been a level 1 expression. Translating it would thus have produced a level 1 λ^U term which, if rebuilt at level 1, would give exactly the same result produced by the term in Example 1.

Example 3 (Basic Macro Declarations).

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

Example 4 (SNDs). Consider the following SML datatype:

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

The SND term $(x, \lambda y.z).(\lambda y.\tilde{}(z\ \langle y\rangle))(\texttt{unC}\ \tilde{}x)$ defines a "monadic-let" macro for this datatype.

$$\begin{aligned}
&\left[\!\!\left[\begin{array}{l}\mathsf{letmac}\ L\ (x, \lambda y.z) = (\lambda y.\tilde{}(z\ \langle y\rangle))(\texttt{unC}\ \tilde{}x)\\ \mathsf{in}\ L_{(*,\lambda *)}\ (\texttt{C}\ 7, \lambda x.\texttt{C}\ (x+x))\end{array}\right]\!\!\right]^1\\
={}&\left\{\begin{array}{l}\tilde{}(\mathsf{letrec}\ L\ x' =\\ \quad \mathsf{let}\ x = \lambda\mathrm{scope}_x^{(x,\lambda y.z)}(x, \lambda y.z).\varPhi_x^{(x,\lambda y.z)}(x')\\ \quad \mathsf{let}\ z = \lambda\mathrm{scope}_z^{(x,\lambda y.z)}(x, \lambda y.z).\varPhi_z^{(x,\lambda y.z)}(x')\\ \quad \mathsf{in}\ \langle[\![(\lambda y.\tilde{}(z\ \langle y\rangle))(\texttt{unC}\ \tilde{}x)]\!]^1\rangle\\ \ \ \mathsf{in}\ \langle[\![L_{(*,\lambda *)}\ (\texttt{C}\ 7, \lambda x.\texttt{C}\ (x+x))]\!]^1\rangle)\end{array}\right.\\
={}&\left\{\begin{array}{l}\tilde{}(\mathsf{letrec}\ L\ x' =\\ \quad \mathsf{let}\ x = \pi_1\ x'\\ \quad \mathsf{let}\ z = \lambda y.((\pi_2\ x')y)\\ \quad \mathsf{in}\ \langle(\lambda y.\tilde{}(z\ \langle y\rangle))(\texttt{unC}\ \tilde{}x)\rangle\\ \ \ \mathsf{in}\ \langle\tilde{}(L\ ([\![\texttt{C}\ 7]\!]^*, [\![\lambda x.\texttt{C}\ (x+x)]\!]^{\lambda *}))\rangle)\end{array}\right.\\
={}&\left\{\begin{array}{l}\tilde{}(\mathsf{letrec}\ L\ x' =\\ \quad \mathsf{let}\ x = \pi_1\ x'\\ \quad \mathsf{let}\ z = \lambda y.((\pi_2\ x')y)\\ \quad \mathsf{in}\ \langle(\lambda y.\tilde{}(z\ \langle y\rangle))(\texttt{unC}\ \tilde{}x)\rangle\\ \ \ \mathsf{in}\ \langle\tilde{}(L\ (\langle\texttt{C}\ 7\rangle, \lambda x.\langle\texttt{C}\ (\tilde{}x + \tilde{}x)\rangle))\rangle)\end{array}\right.
\end{aligned}$$

Evaluating the final term according to the standard λ^U semantics at level 1 yields $(\lambda y.\texttt{C}\ (y+y))\ (\texttt{unC}\ (\texttt{C}\ 7))$.

6 Reasoning about SND Programs

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

Definition 5. *Two SND expressions* e_1 *and* e_2 *are* observationally equivalent *if 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 observationally equivalent level* m λ^U *terms.*

We write $e_1 \approx_m e_2$ when e_1 and e_2 are observationally equivalent SND terms.

To specify the equational theory for SND and show that it has the desired properties, we use the following five sets to categorize the syntax of SND terms after the macro expansion phase has been completed. All are subsets of E_{SND}.

Definition 6.

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

The set V^1_{SND} is a subset of E_{SND}, but it does not allow for escapes, macros, or brackets in terms. The interpretation of SND in λ^U preserves syntactic categories, i.e., $[\![v^m]\!]^m \in V^m_{\lambda^U}$ and $[\![v^p]\!]^{\overline{p}} \in V^0_{\lambda^U}$.

Definition 7 (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:*

$$\begin{aligned}
\pi_i\ (v^0_1, v^0_2) &\rightarrow_\pi\ v^0_i \\
(\lambda x.e^0_1)v^0_2 &\rightarrow_\beta\ e^0_1[x := v^0_2] \\
\textsf{letrec}\ f\ x = e^0_1\ \textsf{in}\ e^0_2 &\rightarrow_{\textsf{rec}}\ e^0_2[f := \lambda x.e^0_1[f := \textsf{letrec}\ f\ x = e^0_1\ \textsf{in}\ f]] \\
(p.e^0_1)_{\overline{p}}\ v^p_2 &\rightarrow_\mu\ e^0_1[v_i = \lambda scope^p_{v_i}(p).\Theta^p_{v_i}(v^p_2)],\quad \{v_i\} = FV(p) \\
\tilde{}\langle v^1 \rangle &\rightarrow_{\textsf{esc}}\ v^1 \\
\textsf{letmac}\ fp\ = e^1_1\ \textsf{in}\ e^1_2 &\rightarrow_{\textsf{mac}} \begin{cases} \tilde{}(\textsf{letrec}\ f\ x = \\ \quad \textsf{let}\ v_i = \lambda scope^p_{v_i}(p).\Phi^p_{v_i}(x) \\ \quad \textsf{in}\ \langle e^1_1 \rangle \\ \quad \textsf{in}\ \langle e^1_2 \rangle)\ \textit{where}\ \{v_i\} = FV(p) \end{cases}
\end{aligned}$$

Here,

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

Note that $\Phi^p_z([\![e^p]\!]^{\overline{p}}) = [\![\Theta^p_z(e^p)]\!]^0$ shows that the use of signatures is implicit in the definition of $\Theta^p_z(e)$.

Theorem 2 (Soundness of SND Reductions).
If $P \vdash^m e_1$ *and* $P \vdash^m e_2$ *then* $e_1 \rightarrow e_2 \Longrightarrow e_1 \approx_m e_2$.

6.1 A 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 equational theory for this language. These equalities will hold for *any* typed variant of SND. To illustrate how such a type system is defined and 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:

$$t \in T_{SND} ::= \mathsf{nat} \mid t * t \mid t \to t \mid \langle t \rangle$$

Here, nat is representative for various base types, $t_1 * t_2$ is a type for pairs comprising values of types t_1 and t_2, $t_1 \to t_2$ 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 the additional condition that any variable name occurs exactly once in any valid context Γ.

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

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 4 presents a type system for SND and various auxiliary judgments needed in defining it.

6.2 Type Safety for SND

Type safety for SND is established by showing that the interpretation maps well-typed SND terms to well-typed λ^U terms (which themselves are known to be type-safe). SND types map unchanged to λ^U types. The translation on type contexts "flattens" 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 interprets each binding in a type context Γ as follows:

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

The translation of SND into λ^U preserves types in the following sense. Suppose $[\![\Gamma]\!]$ is well-defined and y_i are the underlined variables in Γ. 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 λ^U judgment. Similarly, 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 λ^U judgment. Comparing this result to the corresponding one 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 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 $\vdash \overline{p} : t$.

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

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

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

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

$$\frac{\Gamma, p : t_1^0 \vdash^1 e : t_2 \quad \vdash \overline{p} : t_1}{\Gamma \vdash^0 p.e : t_1 \to \langle t_2\rangle}$$

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

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

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

$$\frac{\Gamma \vdash^1 e : t_1}{\Gamma \vdash^0 \langle e\rangle : \langle t_1\rangle} \qquad \frac{\Gamma \vdash^0 e : \langle t_1\rangle}{\Gamma \vdash^1 \tilde{}e : t_1} \qquad \frac{\Gamma \vdash^1 e_1 : t_1}{\Gamma \vdash^* e_1 : \langle t_1\rangle} \qquad \frac{\Gamma \vdash^0 e : t_1}{\Gamma \vdash^{\tilde{}*} e : t_1}$$

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

Fig. 4. Type System for SND

Theorem 3 (Type Safety). *If $\vdash^m e : t$ is a valid SND judgment, then translating e to λ^U yields a well-typed λ^U program, and executing that program does not generate any λ^U run-time errors.*

7 Embedding of MacroML in SND

Embedding MacroML into SND requires a minor modification to the original definition of how MacroML is interpreted in a multi-stage language. In particular, MacroML macros have exactly three arguments (representative of the three possible kinds of arguments: early parameters, regular parameters, and new binding constructs). In the original definition of MacroML, these arguments are taken to be curried. It simplifies the embedding to modify the original definition to treat these three arguments as a tuple.

The type system of MacroML is virtually unchanged, and is reproduced in Figure 5. The modified interpretation of MacroML in λ^U is presented in Figure 7. The embedding of MacroML into SND is given in Figure 6.

$$\frac{x : t^m \in \varGamma}{\varSigma; \varDelta; \varPi; \varGamma \vdash^m x : t} \qquad \frac{x : t \in \varPi \text{ or } \underline{x} : t \in \varPi}{\varSigma; \varDelta; \varPi; \varGamma \vdash^1 x : t} \qquad \frac{x_2 : [x_1 : t_1]t_2 \in \varDelta \quad x_1 : t_1^1 \in \varGamma}{\varSigma; \varDelta; \varPi; \varGamma \vdash^1 x_2 : t_2}$$

$$\frac{\varSigma; \varDelta; \varPi; \varGamma, x : t_1^m \vdash^m e : t_2}{\varSigma; \varDelta; \varPi; \varGamma \vdash^m \lambda x.e : t_1 \to t_2} \qquad \frac{\varSigma; \varDelta; \varPi; \varGamma \vdash^m e_1 : t_2 \to t \quad \varSigma; \varDelta; \varPi; \varGamma \vdash^m e_2 : t_2}{\varSigma; \varDelta; \varPi; \varGamma \vdash^m e_1 e_2 : t}$$

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

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

$$\frac{\varSigma; \varDelta; \varPi; \varGamma \vdash^0 e : \langle t \rangle}{\varSigma; \varDelta; \varPi; \varGamma \vdash^1 \tilde{}e : t}$$

Fig. 5. MacroML Type System (with underlines)

Theorem 4 (Embedding). *Translating any well-formed MacroML term into SND and then into λ^U is equivalent to translating the MacroML term directly into λ^U. That is, If* $\varSigma; \varDelta; \{x_i : t_i\}; \varGamma \vdash^m e : t$ *is a valid MacroML judgment, then* $[\![[\![\varSigma; \varDelta; \{x_i : t_i\}; \varGamma \vdash^m e : t]\!]^M]\!]^m [x_i := \tilde{}x_i] \approx_m [\![\varSigma; \varDelta; \{x_i : t_i\}; \varGamma \vdash^m e : t]\!]$.

8 Conclusions

Previous work demonstrated that the "macros as multi-stage computations" approach is instrumental for developing and verifying type systems for expressive macro languages. 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 λ^U 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.

Lambda Terms

$$\frac{x : t^m \in \mathit{\Gamma}}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^m x : t \rrbracket^M = x} \qquad \frac{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma}, x : t_1^m \vdash^m e : t_2 \rrbracket^M = e'}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^m \lambda x.e : t_1 \to t_2 \rrbracket^M = \lambda x.e'}$$

$$\frac{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^m e_1 : t_2 \to t \rrbracket^M = e_1' \quad \llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^m e_2 : t_2 \rrbracket^M = e_2'}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^m e_1 e_2 : t \rrbracket^M = e_1' e_2'}$$

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

Macros

$$\frac{\underline{x} : t \in \mathit{\Pi}}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^1 x : t \rrbracket^M = x} \qquad \frac{x : t \in \mathit{\Pi}}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^1 x : t \rrbracket^M = \tilde{}x}$$

$$\frac{x_2 : [x_1 : t_1]t_2 \in \mathit{\Delta} \quad x_1 : t_1^1 \in \mathit{\Gamma}}{\llbracket \mathit{\Sigma}; \mathit{\Delta}; \mathit{\Pi}; \mathit{\Gamma} \vdash^1 x_2 : t_2 \rrbracket^M = \tilde{}(x_2\ \langle x_1 \rangle)}$$

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

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

Code Objects

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

Fig. 6. Translation from MacroML to SND

Previous work on MacroML indicated a need for making explicit the escape and bracket constructs in the language, so that unfolding recursive macros could 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, suggest that macro languages may naturally and usefully be viewed as conservative extensions of multi-stage (or at least two-level) languages.

Acknowledgments Amr Sabry was the first to notice that alpha-conversion in the context of MacroML's `letmac` could be problematic. We thank the anonymous referees for their helpful and detailed comments, some of which we were not able to address fully due to space constraints. Stephan Ellner proofread a draft of the paper.

References

1. Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed. 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 Satellite Workshop on Intuitionistic Modal Logics and Applications* (1999).
3. Cardelli, L. Phase distinctions in type theory. (Unpublished manuscript.) Available online from http://www.luca.demon.co.uk/Bibliography.html, 1988.
4. Davies, R. A temporal-logic approach to binding-time analysis. In *Symposium on Logic in Computer Science* (1996), IEEE Computer Society Press, pp. 184–195.
5. Filinski, A. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming* (1999), vol. 1702 of *LNCS*, pp. 378–395.
6. Filinski, A. Normalization by evaluation for the computational lambda-calculus. In *Typed Lambda Calculi and Applications: 5th International Conference* (2001), vol. 2044 of *LNCS*, pp. 151–165.
7. Ganz, S., Sabry, A., and Taha, W. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *International Conference on Functional Programming* (2001), ACM.
8. Griffin, T. G. Notational definitions — a formal account. In *Proceedings of the Third Symposium on Logic in Computer Science* (1988).
9. Michaylov, S., and Pfenning, F. Natural semantics and some of its meta-theory in Elf. In *Extensions of Logic Programming* (1992), L. Hallnäs, Ed., LNCS.
10. Moggi, E. Functor categories and two-level languages. In *Foundations of Software Science and Computation Structures* (1998), vol. 1378 of *LNCS*.
11. Moggi, E. A monadic multi-stage metalanguage. In *Foundations of Software Science and Computation Structures* (2003), vol. 2620 of *LNCS*.
12. Moggi, E., Taha, W., Benaissa, Z. E.-A., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming* (1999), vol. 1576 of *LNCS*, pp. 193–207.
13. 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.
14. Sheard, T., and Peyton-Jones, S. Template meta-programming for Haskell. In *Proc. of the Workshop on Haskell* (2002), ACM, pp. 1–16.
15. Steele, Jr., G. L., and Gabriel, R. P. The evolution of LISP. In *Proceedings of the Conference on History of Programming Languages* (1993), R. L. Wexelblat, Ed., vol. 28(3) of *ACM Sigplan Notices*, ACM Press, pp. 231–270.
16. Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [13].
17. 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* (2000), ACM Press.
18. Taha, W., Benaissa, Z.-E.-A., and Sheard, T. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming* (1998), vol. 1443 of *LNCS*, pp. 918–929.
19. Taha, W., and Nielsen, M. F. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)* (New Orleans, 2003).

Type Contexts

$$
\begin{aligned}
\llbracket \emptyset \rrbracket &= [\,] \\
\llbracket \Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5 \rrbracket &= \llbracket \Sigma \rrbracket, f : (t_1 * (\langle t_2 \rangle * (\langle t_3 \rangle \rightarrow \langle t_4 \rangle)) \rightarrow \langle t_5 \rangle)^0 \\
\llbracket \Delta, x_2 : [x_1 : t_1]t_2 \rrbracket &= \llbracket \Delta \rrbracket, x_2 : (\langle t_1 \rangle \rightarrow \langle t_2 \rangle)^0 \\
\llbracket \Pi, x : t \rrbracket &= \llbracket \Pi \rrbracket, x : \langle t \rangle^0 \\
\llbracket \Pi, \underline{x} : t \rrbracket &= \llbracket \Pi \rrbracket, x : \langle t \rangle^0
\end{aligned}
$$

Lambda Terms

$$
\frac{x : t^m \in \Gamma}{\llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m x : t \rrbracket = x}
\qquad
\frac{\llbracket \Sigma; \Delta; \Pi; \Gamma, x : t_1^m \vdash^m e : t_2 \rrbracket = e'}{\llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m \lambda x.e : t_1 \rightarrow t_2 \rrbracket = \lambda x.e'}
$$

$$
\frac{\llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 : t_2 \rightarrow t \rrbracket = e_1' \quad \llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m e_2 : t_2 \rrbracket = e_2'}{\llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 e_2 : t \rrbracket = e_1' e_2'}
$$

$$
\frac{\begin{array}{l}\llbracket \Sigma; \Delta; \Pi; \Gamma, f : (t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t)^m, x_1 : t_1^m, x_2 : t_2^m, x_3 : t_3^m \vdash^m e_1 : t \rrbracket = e_1' \\ \llbracket \Sigma; \Delta; \Pi; \Gamma, f : (t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t)^m \vdash^m e_2 : t_4 \rrbracket = e_2' \end{array}}{\llbracket \Sigma; \Delta; \Pi; \Gamma \vdash^m \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 : t_4 \rrbracket = \mathsf{letrec}\ f\ x_1 = \lambda x_2.\lambda x_3.e_1'\ \mathsf{in}\ e_2'}
$$

Macros

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

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

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

Code Objects

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

Fig. 7. Modified Interpretation of MacroML in λ^U