Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML

Steve Ganz [*]
Computer Science Department
Indiana University
sganz@cs.indiana.edu

Amr Sabry[†]
Computer Science Department
Indiana University
sabry@cs.indiana.edu

Walid Taha[‡]
Department of Computer Science
Yale University
taha@cs.yale.edu

ABSTRACT

Macros have traditionally been viewed as operations on syntax trees or even on plain strings. This view makes macros seem *ad hoc*, and is at odds with two desirable features of contemporary typed functional languages: static typing and static scoping. At a deeper level, there is a need for a simple, usable semantics for macros. This paper argues that these problems can be addressed by formally viewing macros as multi-stage computations. This view eliminates the need for suspicious freshness conditions and tests on variable names, and provides a compositional interpretation that can serve as a basis for designing a sound type system for languages supporting macros, or even for compilation.

To illustrate the proposed approach, we develop and present MacroML, an extension of ML that supports inlining, recursive macros, and the definition of new binding constructs. The latter is subtle, and is the most novel addition in a statically typed setting. The semantics of a core subset of MacroML is given by an interpretation into MetaML, a statically-typed multi-stage programming language. It is then easy to show that MacroML is stage- and type-safe: macro expansion does not depend on run-time evaluation, and neither "goes wrong" in the usual sense.

1. INTRODUCTION

Most real programming language *implementations* provide a macro facility that can be used to improve either performance or expressiveness, or both. In the first case, macros are usually used for inlining or unfolding particular function calls. In the second case, macros are usually used to define new language constructs or shorthands. Many tasks can be achieved using such macro systems, including conditional compilation, configuration of applications to particular environments, templates for parameterized computations, and even the implementation of domain-specific languages. Yet macros are not part of the standards for the mainstream statically-typed functional languages such as ML and Haskell.

*This work was supported in part by the National Science Foundation under grant CDA-9312614.

†This material is based upon work supported by the National Science Foundation under Grant No. CCR-0196063.

‡Funded by subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

So, why are they ignored?

Often, macros are considered to be either an implementation detail (and therefore not interesting) or, a form of black magic (and therefore should be kept at bay). Both these stands are unfounded. First, it is a mistake to give macros (or even inlining pragmas) the status of a compiler directive that the implementation may or may not take into account: macros affect the semantics of programs (see Section 2). Second, the absence of a macro facility almost invariably forces programmers to resort to *ad hoc* solutions to achieve the same functionality.

There are also technical difficulties: macros are hard to specify from first principles. Macro designers often find themselves forced to describe macros at the level of program text or syntax trees, and to address the inevitable hygiene problems and scoping problems using `gensym` (or freshness conditions) and many unintuitive equality and inequality tests on variables names [19]. Not only are such low-level specifications hard to communicate (making macros acquire *the appearance* of being unsystematic), they are also at odds with static typing: if variable names and their binding relationships are not known until *after* macro expansion, it becomes hard (if not impossible) to type-check macros before expansion and evaluation start.

This paper argues that macro systems can be viewed formally and usefully as multi-stage computations. Multi-stage programming languages (including two-level languages [28, 16], multi-level languages [13, 14, 15, 8, 7], and MetaML [41, 39, 5]) have been developed precisely to provide precise and usable models of such computations that occur in multiple distinct stages. Over the last few years, the study of MetaML and related systems has not only resulted in a good understanding of the types and semantics of multi-stage systems, but has also solved, once and for all, some of the hard problems (freshness conditions, typing) in macro systems.

Formalizing macros as multi-stage computations also emphasizes that the technical problems associated with macros are genuine: specifying the denotational semantics of macros involves the same advanced tools as the denotational semantics for two-level, multi-level, and multi-stage languages (such as functor-categories or topoi [24, 25, 4]). A denotational semantics has particular relevance to realistic compilers, which invariably involve a translation phase. A compositional (denotational) semantics is generally one of the most helpful kinds of semantics in developing and verifying such compilers [30].

While this paper demonstrates that MetaML is a good meta-language for *defining* macros, MetaML is not the ideal language for *writing* macros: it does not have support for defining new binding constructs. In addition to dictating and controlling inlining, macros are often used to abstract common syntactic patterns in programs. In a language with higher-order procedures, especially a lazy language, some of these abstractions can be expressed (maybe with a loss of efficiency) using functions. But many of the syntactic patterns over which one wants to abstract would need to bind variables. Constructs that bind variables are not directly expressible using functions. For example, overloading an existing binding construct such as the `do`-notation of Haskell to allow recursive bindings cannot be expressed using functions, and requires a change to the compiler [10].

MacroML: This paper presents an expressive, typed language that supports generative macros. This language, called MacroML, is defined by an interpretation into MetaML, and can express (both simple and recursive) inlining and the definition of new binding constructs. A key design goal for MacroML is that it be a *conservative extension* [11] of ML. This implies that its type system should include all well-typed ML programs. It should also not break the reasoning principles for ML programs, such as α and β_v conversion. We also want the language to remain statically typable. Given that our goal is a conservative extension of ML, there are some notable points about what MacroML is designed *not* to do:

- MacroML does not blur the distinction between programs and data. Although many applications naturally view programs as data, there is a fundamental distinction between the two. While both programs and data can be represented (using, *e.g.*, natural numbers, S-expressions), *the notions of equality associated with each one (syntactic and semantic equality, respectively) cannot and should not be mixed.* Furthermore, internalizing the two notions into a language, while still enforcing the distinction between the two is non-trivial [22, 44, 39]. For better or worse, it is relatively easy to pick one *or* the other, that is, to either have syntactic equality or semantic quality everywhere. In MacroML, we choose to allow only semantic equality in the language and we avoid introducing syntactic equality (on programs) by not introducing *any* reflective or code-inspection capabilities into the language.
- MacroML does not introduce accidental dynamic scoping and/or variable capture. These problems generally arise from an overly simplistic view of programs as data, such as in early LISP systems or in C. The Scheme community has had a substantial role in recognizing and addressing this problem and promoting the notion of hygienic macro expansion [19, 9]. More recently, there have been more sophisticated proposals, like higher-order abstract syntax (HOAS) [33, 21, 17, 12], and FreshML [34]. The key contribution of all these proposals is to provide a means to express the fact that "programs are not *just* data."
- MacroML does not allow macros that inspect or take apart code (*i.e.*, *analytic macros*). This restriction seems necessary to maintain static typing. Instead all macros in MacroML are limited to constructing new code and combining code fragments (*i.e.*, *generative macros*). This paper presents a number of examples that suggest that many useful tasks can be accomplished using such statically typed generative macros.

Especially the first point is inspired by multi-stage language, but to an extent, so are the other two. A key issue that arises in the presence of macros that define new binding constructs is the handling of α conversion. While it is not clear how this problem can be addressed in the untyped setting, it is also addressed by the type system. As such MacroML tries to achieve a balance between being an expressive macro system and being a macro system that we can reason about.

Organization: Section 2 introduces MacroML by a series of motivating examples, and discusses the issue of alpha equivalence in the presence of macros that can define new binding constructs. Section 3 reviews the MetaML syntax, type system, and semantics. Section 4 presents the main technical contribution: a compositional interpretation of Core MacroML into MetaML that provides a semantics that is both executable and reasonably easy to communicate. Neither the MacroML language nor the translation use any operations nor side-conditions to generate fresh names. Instead this is relegated to the semantics of the target language of the translation. The target language itself, MetaML, has an operational semantics defined using nothing but the standard notion of substitution. The translation is shown to produce only well-typed MetaML terms thus providing a type safety result for MacroML. Section 5 considers several extensions to Core MacroML and discusses implementation issues. Sections 6 and 7 discuss related work and conclude.

2. MACROML BY EXAMPLE

In this section, we use a sequence of examples to introduce the basic issues motivating and governing our design of MacroML. Each example is followed by a summary of the basic semantic concerns that it raises.

2.1 Simple Inlining: A First Attempt

Consider the following code excerpt, where the functions `iterate` and `shift_left` have the expected meaning:

```
let val word_size = 8
in ... iterate shift_left word_size ... end
```

Here, `word_size` is used purely for reasons of clarity and maintainability in the source code, and most implementations are likely to inline it producing:

```
... iterate shift_left 8 ...
```

But in a general situation where `word_size` is bound to a more complicated expression like `x+4` or an expression whose evaluation might have side-effects like `1/x`, the situation is more delicate. Some compilers might inline `x+4` and some might not. And no compiler is at liberty to inline expressions with effects: this is clear in a language like ML, but it is also the case in "pure" languages like Haskell where compilers must restrict inlining when dealing with built-in monadic effects [36, 1]. Compilers also cannot be left to inline or not inline at will when we care about resource behavior [27].

Since inlining affects not only the performance but also the semantics of ML programs, we elevate it to a full language construct with concrete syntax, typing rules, and formal semantics. In MacroML, programmers can *require* inlining of an expression using a new variant of `let`-expressions called `let mac`. For example, the fragment:

```
let mac word_size = raise Unknown_size
in ... iterate shift_left word_size ... end
```

dictates that, even though `raise Unknown_size` is not an ML value, we want it inlined, producing:

```
... iterate shift_left (raise Unknown_size) ...
```

Semantics: The semantics of this kind of inlining is simply the standard capture-avoiding substitution of a variable by an expression [6, 2]. Unfortunately, while this is a good example of the "essence of inlining," introducing inlining in this fashion (through the mere occurrence of a variable) can interfere with established reasoning principles for call-by-value (CBV) languages. For example, in a standard CBV calculus [35], `K 5 x` is observationally equivalent to `5` since variables are values. But in the context:

```
let mac x = raise Error in ... end
```

these two terms behave differently, because, contrary to the usual assumption about CBV variables, `x` is replaced by a non-value. To retain the established reasoning principles of CBV languages, we restrict all uses of macros in MacroML to be syntactically non-values (*e.g.*, applications, `let`-expressions, etc).

2.2 Functional Inlining

Macros that take arguments are just as useful if not more useful than simple inlining. Their form (as applications) also provides the added advantage that they do not interfere with the established semantics of CBV languages. Consider:

```
mac $ e = fn x => e
mac ? e = e ()
```

The declared operators `$` and `?`, read "delay" and "force," respectively, implement a simple variant of Okasaki's proposal for suspensions [29]. There are two notable features about this example. First, we cannot define `$` as a function since the evaluation of `$ e'` should not allow the premature evaluation of `e'`. Thus, this is a genuinely useful application of a macro system. Second, macro expansion should not allow the binding occurrence of `x` in the macro definition to accidently capture free occurrences of `x` in macro arguments.

Semantics: The semantics of functional inlining involves two substitutions. First, the argument of the macro application is substituted into the body. Given that we are using the standard notion of substitution, the variable `x` in the above example cannot occur in the expression bound to the variable `e`. Second, the resulting macro body is substituted back into the context of application, again using the standard notion of substitution. In our example, the expression `fn x => $ (tl x)` expands to `fn x => fn x' => tl x` where `x'` is a freshly generated name (with a base name `x` only to hint to its source).

This example demonstrates another key feature of macro systems: because macro calls can occur under binders, the semantics *requires* evaluation under binders. This is significant because evaluation under a binder generally involves manipulating *open code* which is significantly more complicated [41, 26, 38] than the restriction to dealing with only closed terms imposed by most semantic specifications and implementations of languages (whether CBV, CBN, or call-by-need).

2.3 Recursive Macros

What if we wish to perform more computations during macro expansion? Consider the classic power function `pow`:

```
let fun pow n x =
  if n = 0 then 1 else x * (pow (n-1) x)
in pow (2*3) (5+6) end
```

If we replace the `fun` keyword by `mac`, macro expansion goes into an infinite loop, which is probably not the desired behavior. What happens? Intuitively, the macro call `pow (2*3) (5+6)` expands into :

```
if 2*3 = 0 then 1 else (5+6) * (pow (2*3-1) (5+6))
```

which itself expands into :

```
if 2*3 = 0 then 1
else (5+6) * (if (2*3-1) = 0 then 1
              else (5+6) * (pow ((2*3-1)-1) (5+6)))
```

and the expansion goes on indefinitely. Macro expansion can only terminate if the `if`-expression is evaluated *during* expansion, and not reconstructed as part of the result. To require the execution of the `if` expression during macro expansion, we must explicitly indicate that `n` is an *early* parameter, rather than a regular macro parameter (which we call *late*), and annotate the term to distinguish among early and late computations. The intended `pow` macro can now be written as:

```
let mac pow ~n x =
  ~(if n = 0 then <1> else <x * ~(pow (n-1) x)>)
in pow ~(2*3) (5+6) end
```

The two constructs *escape* ~*e* and *brackets* ⟨*e*⟩ are borrowed directly from MetaML, and work as follows: the first construct "escapes" from the "macro expansion mode" to regular ML evaluation to perform a computation; the second construct interrupts regular evaluation to return a result to the "macro expansion mode." The expansion of the macro call above now yields:

```
(5+6) * (5+6) * (5+6) * (5+6) * (5+6) * (5+6) * 1
```

Semantics: The need for introducing the brackets and escape constructs is directly related to the need to have a well-specified order for evaluating various sub-expressions. In particular, with recursion, it becomes clear that there are two different kinds of computations: early ones and late ones. The need to intermix these two kinds of computation is what requires a more substantial type system than usual. In particular, we must enforce what is called *congruence* in the partial evaluation literature: a well-formed multi-stage computation should not contain an early computation that depends on the result of a late computation [18]. A simple example of the kind of program that the type system has to reject is:

```
let mac f b n = ~(if b=0 then <n> else <n+1>)
in fn a => fn m => f a m end
```

The macro argument `a` is needed during macro expansion but, being a run-time variable, it is unbound at that time.

2.4 Defining New Binding Constructs

We now come to one of the most novel features of MacroML: the ability to define new binding constructs in the typed setting. Let us say that we are using the macros `$` and `?` for suspensions to implement a notion of a computation [23]. We define a suitable monadic-`let` for this setting as follows:

```
mac (let seq x = e1  in e2  end) =
   $(let val x = ?e1 in ?e2 end)
```

The definition introduces a new binding construct `let seq` which expands to the core binding construct `let val`. For example,

```
let seq y = f $7 in g y end
```

expands to:

```
$(let val x = ?(f $7) in ?(g x) end)
```

Semantics: A key insight behind this aspect of our proposal is to allow the user to only define new binding constructs that follow the patterns of existing binding constructs, such as lambda abstractions, value declarations, and recursive declarations. *In these binding constructs every occurrence of a variable can be immediately identified as either a binding occurrence or a bound occurrence.* The semantics of our proposal is designed to reflect this distinction. But even when this distinction is taken into account, α equivalence is still subtle. For example, the definition above cannot be written into:

```
mac (let seq x = e1  in e2  end) =
   $(let val y = ?e1 in ?e2 end)
```

with the justification that "`y` is a binding occurrence, and it does not occur in the body `?e2`." The problem here is that there are in fact two different binding occurrences of the variable `x`, and each one of them is of a different nature. In essence, the first one (in the parameter of the macro) says that "there is a variable, let's call it `x`, which can occur free in the expression bound to `e2`". Because of this, the use of the variable name `x` in the second declaration now has special meaning. In essence, the second declaration now says "use `x` locally as a normally variable name, but make sure that it is treated in the output of the macro as the binding occurrence for the `x` in `e2`."

The type system for MacroML addresses this issue through two mechanisms: first, special type environments are used to keep track of the declarations of macro parameters, and most importantly, the *bindee* parameters like `e2` above. Second the type of these bindee parameters will explicitly carry around the name of the *binder* parameter `x`, which comes from the first declaration of `x`. The second declaration of `x` itself is in fact a completely normal declaration. With this typing information, it is possibly to reject the local replacement of `x` into `y` as above. It is important to note the difference between this mechanism and the classic "accidental dynamic scoping:" *the dependency on a "free" variable is reflected explicitly in the type.* Essentially the same principle underlies the recent proposal for implicit parameters [20]. We know that the type system provides an adequate solution to the problem of α conversion in the source program because the type system guarantees that well-typed MacroML programs can be translated to MetaML programs, and in the latter, α renaming is completely standard, even in the untyped setting.

At this point, the reader may wonder "how are variables passed around in MacroML?" We return shortly to this question in Section 4.

3. MULTI-STAGE LANGUAGES

Macro systems introduce a stage of computation that should happen before the traditional stage of program execution. Early computations during this new stage include macro expansion as well as various other traditional computations (*e.g.*, conditionals, applications, etc).

As mentioned in the introduction, multi-level languages have been developed to model such kinds of staged computation.

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

$$\frac{\begin{array}{l}\Gamma, f : (t_1 \to t_2 \to t_3 \to t)^n, x_1 : t_1^n, x_2 : t_2^n, x_3 : t_3^n \vdash^n e_1 : t \\ \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^n \vdash^n e_2 : t_4\end{array}}{\Gamma \vdash^n \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 : t_4} \qquad \frac{\Gamma \vdash^{n+1} e : t}{\Gamma \vdash^n \langle e \rangle : \langle t \rangle} \qquad \frac{\Gamma \vdash^n e : \langle t \rangle}{\Gamma \vdash^{n+1} \tilde{\ } e : t} \qquad \frac{\Gamma^+ \vdash^n e : \langle t \rangle}{\Gamma \vdash^n \mathsf{run}\ e : t}$$

Figure 1: MetaML Type System

Multi-level languages offer constructs for building and combining code, often in a typed setting. Multi-stage languages are multi-level languages that provide the user with a means of executing the generated code. The notion of "code" described here is an abstract one. For example, it has been proven that beta-reductions are sound inside this notion of code [39]. This means that code in such systems is never inspected (nor observed) syntactically. An alternative, equally valid way of thinking about these languages, therefore, is that they provide *fine control over the evaluation order* [41, 40, 37].

A premier example of a statically-typed, functional, multi-stage language is MetaML. In addition to the normal constructs of a functional language, MetaML also has three constructs for building, efficiently combining, and executing code. These three constructs are *brackets* $\langle e \rangle$, *escape* $\tilde{\ } e$, and *run* run e. For the purposes of our study here, we use the following small MetaML language as our multi-stage language:

$$\begin{array}{rcl} e \in E_{Meta} & ::= & x \mid \lambda x.e \mid e\ e \\ & \mid & \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 \\ & \mid & \langle e \rangle \mid \tilde{\ } e \mid \mathsf{run}\ e \end{array}$$

We have restricted the number of arguments of recursive functions to three, only for simplicity.

We present a type system for this language with the following types:

$$t \in T_{Meta} \quad ::= \quad \mathsf{nat} \mid t \to t \mid \langle t \rangle$$

Here nat is the type for natural numbers, used as an example of various base types. In this paper, we have omitted giving constructs that work on this type because their treatment is trivial, and space is limited. Function types as usual have the form $t \to t$. The MetaML code type is denoted by $\langle t \rangle$. In this section we present a sound type system for MetaML. The soundness of this type system is studied and established elsewhere [40, 26, 38]. While this type system is not the most expressive type system available for MetaML (see for example [26, 38, 5]), it is simple and sufficient for our purposes here.

The judgments of the type system have the form $\Gamma \vdash^n e : t$, where n is a natural number called the *level* of a term. The role of the notion of level is explained below where we consider the type rules and the semantics for brackets and escape. The 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 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 1. The first four rules of the type system are standard, except that the level n of each term is passed around everywhere. Note that in the rule for lambda (and recursive functions), we take the current level and use it as the level of the bound variable when we add it to the environment.

The rule for brackets says $\langle e \rangle$ can have type $\langle t \rangle$ when e has type t. In addition, e must be typed at level $n+1$, where n is the level for $\langle e \rangle$. The level parameter is therefore counting the number of "surrounding" brackets. The rule for escape does basically the converse. Note, therefore, that escapes can only occur at level 1 and higher. Escapes are supposed to "undo" the effect of brackets.

Finally, 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. However, we must be careful to note that the term being run must be typed under the environment Γ^+, rather than simply Γ. We define Γ^+ as having the same variables and corresponding types as Γ, but with each level incremented by 1. Without this (rather subtle) adjustment the type system is unsafe [40, 26, 38].

Figure 2 defines the big-step semantics for MetaML. There are a number of reasons why the big-step semantics for MetaML [26, 40] is an instructive model for the formal study of multi-stage computation: first, by making "evaluation under lambda" explicit, this semantics makes it easy to illustrate how a multi-stage computation often violates one of the basic assumptions of many works on programming language semantics, namely, the restriction to closed terms. Second, by using just the standard notion of substitution [2], this semantics captures the *essence* of static scoping, and there is no need for using additional machinery to perform renaming at run-time.

The big-step semantics for MetaML is a family of partial functions $\_ \overset{n}{\hookrightarrow} \_ : E_{Meta} \to E_{Meta}$ from expressions to "answers," indexed by a natural number n (the level). 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 they are evaluated by rebuilding the expression they surround 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. Thus rebuilding simply traverses a term until

$$\frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} e_3 \quad e[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_1\ e_2 \overset{0}{\hookrightarrow} e_4} \qquad \frac{F \equiv \lambda x_1.\lambda x_2.\lambda x_3.e_1[f := \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ f] \quad e_2[f := F] \overset{0}{\hookrightarrow} e_3}{\mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} e}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3} \qquad \frac{}{x \overset{n+1}{\hookrightarrow} x} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+1}{\hookrightarrow} \lambda x.e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+1}{\hookrightarrow} e_3\ e_4}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} e_3 \quad e_2 \overset{n+1}{\hookrightarrow} e_4}{\mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 \overset{n+1}{\hookrightarrow} \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_3\ \mathsf{in}\ e_4}$$

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

Figure 2: MetaML Big-Step Semantics

a level 1 escape is encountered, at which point normal (level 0) evaluation function is invoked. The escaped expression must yield a bracketed expression, and then the expression itself is returned.

Interesting examples of MetaML programs can be found in the literature [41, 38, 37]. For the purposes of this paper, we focus on illustrating how three kinds of computation can be achieved using MetaML:

Evaluation Under Lambda: Consider the term $\lambda xy.(\lambda z.z)\ x$ and let us say that we are interested in eliminating the inner application. In both CBV and CBN, evaluation only works on closed terms, and therefore, never "goes under lambda." With MetaML it is possible to force the inner computation by rewriting the expression as $\mathsf{run}\ \langle \lambda xy.\tilde{}((\lambda z.z)\ \langle x \rangle)\rangle$, and then evaluating it. The result is the desired term: $\lambda xy.x$. We use such a pattern of run, brackets, and escape in our interpretation of the macro language to ensure that computations are performed in the desired order.

Substitution: Consider the term $(\lambda x.f\ x\ x)\ (g\ y)$. Can we force the application to be done first, producing $f\ (g\ y)\ (g\ y)$? This operation is not expressible in CBV or CBN *evaluation* semantics, but is expressible in MetaML by annotating the term as follows: $\mathsf{run}\ ((\lambda x.\langle f\ \tilde{}x\ \tilde{}x\rangle)\ \langle g\ y\rangle)$.

Renaming: While it may be obvious to experts, it seems not widely known that simply using the standard notion of substitution in defining the semantics of a language like MetaML is sufficient for providing the correct treatment of free and bound variables everywhere. In MetaML, there is never any accidental variable capture, and there is never any need to express any freshness conditions or to use a `gensym`-like operation. Our semantics for MacroML is simple, because we build on the fact that using the standard notion of substitution in the MetaML semantics really means that everything is taken care of.

4. CORE MACROML

We are now at a point where we can precisely define and interpret our macro language. We present the syntax, type system, and semantics of the core of MacroML (called Core MacroML). The language has the usual expressions for a CBV language, augmented with the previously-motivated let mac-construct for defining macros, and the $\tilde{}e$ and $\langle e \rangle$ used to control recursive inlining:

$$\begin{array}{rcl} e \in E_{Macro} & ::= & x \mid \lambda x.e \mid e\ e \\ & \mid & \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 \\ & \mid & \mathsf{letmac}\ f(\tilde{}x_0, x_1, \lambda x_2.x_3)\ = e_1\ \mathsf{in}\ e_2 \\ & \mid & f(e_1, e_2, \lambda x.e) \mid \langle e \rangle \mid \tilde{}e \end{array}$$

For clarity, all macros in Core MacroML have exactly three parameters representative of the three *kinds* of possible parameters in MacroML. Restricting ourselves to macros with exactly three parameters allows us to avoid substantial administrative detail in Core MacroML. The three kinds of macro arguments are as follows:

1. x_0, as indicated by the preceding $\tilde{}$, is an *early* parameter, which can be used during macro expansion,

2. x_1 is a *late* parameter, which might appear in the output of the expansion, and

3. $\lambda x_2.x_3$ defines a *binder/bindee* pair of parameters. The two variables x_2 and x_3 are bound variables in the scope of the macro definition but they can only be used in rather special ways enforced by the type system. The variable x_2 must be bound *again* using a regular binding construct before x_3 can be used. The variable x_3 can only be used in the scope of x_2. A legal use of such a binder/bindee pair is:

   $$\mathsf{letmac}\ f(\_, \_, \lambda x.y)\ = \lambda a.\lambda x.a + x + y\ \mathsf{in}\ \ldots$$

   The x in the macro declaration binds the *two* occurrences of x in the macro definition! All three occurrences of x could be renamed to z without changing the meaning. The semantics would, however, be changed if we only change the inner term to $\lambda z.a + z + y$,

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

$$\frac{\Sigma; \Delta; \Pi; \Gamma, x : t_1^m \vdash^m e : t_2}{\Sigma; \Delta; \Pi; \Gamma \vdash^m \lambda x.e : t_1 \to t_2} \qquad \frac{\Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 : t_2 \to t \quad \Sigma; \Delta; \Pi; \Gamma \vdash^m e_2 : t_2}{\Sigma; \Delta; \Pi; \Gamma \vdash^m e_1 e_2 : t}$$

$$\frac{\begin{array}{l}\Sigma; \Delta; \Pi; \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m, x_1 : t_1^m, x_2 : t_2^m, x_3 : t_3^m \vdash^m e_1 : t \\ \Sigma; \Delta; \Pi; \Gamma, f : (t_1 \to t_2 \to t_3 \to t)^m \vdash^m e_2 : t_4\end{array}}{\Sigma; \Delta; \Pi; \Gamma \vdash^m \mathsf{letrec}\ f\ x_1\ x_2\ x_3 = e_1\ \mathsf{in}\ e_2 : t_4}$$

$$\frac{\begin{array}{l}\Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta, x_2 : [x : t_3]t_4; \Pi, x_1 : t_2; \Gamma, x_0 : t_1^0 \vdash^1 e_1 : t_5 \\ \Sigma, f : (t_1, t_2, [t_3]t_4) \Rightarrow t_5; \Delta; \Pi; \Gamma \vdash^1 e_2 : t\end{array}}{\Sigma; \Delta; \Pi; \Gamma \vdash^1 \mathsf{letmac}\ f(\tilde{}x_0, x_1, \lambda x.x_2)\ = e_1\ \mathsf{in}\ e_2 : t}$$

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

Figure 3: MacroML Type System

because we would be returning a result that could have an unbound variable (that was bound to x) in a subterm (that was bound to y). The binder/bindee parameter illustrates how defining new binding constructs works in MacroML. For Core MacroML we have picked the simplest binding construct in the language, namely lambda abstraction $\lambda x.e$. The other binding constructs follow naturally.

The application of a macro $f(e_1, e_2, \lambda x.e)$ also takes exactly three arguments: the first is an early argument, the second is a late argument, and the third is a binder/bindee argument. The binder/bindee argument explains the core of our treatment of new binding constructs: it must be clear what variables are free in what sub-expressions, and both must always be passed together. Note that a binder/bindee argument must have the right form for the binding structure (in this case, lambda). For example, the application $f((), (), \lambda z.z + a)$ of the macro defined above expands to

$$\lambda a'.\lambda x.a' + x + (x + a).$$

4.1 Typing Core MacroML

The types of MacroML are the same as MetaML:

$$t \in T_{Macro} \quad ::= \quad \mathsf{nat} \mid t \to t \mid \langle t \rangle$$

The type system, however, is more involved. Typing judgments have the form $\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t$ where m is the level of a term. We restrict the levels here to 0 (representing early computations) and 1 (representing late computations). The environments have the following roles (and definitions):

- Σ the *macro environment.* It keeps track of the various macros that have been declared. These bindings are of the form $(t_1, t_2, [t_3]t_4) \Rightarrow t_5$. In this binding, the tuple provides information about the three standardized parameters. Note that we write $[t_3]t_4$ to describe the binder/bindee argument. Intuitively, the binder/bindee pair is a pair of a "bound" variable declared to be of type t_3 and an expression of type t_4, which could contain a free occurrence of the "bound" variable. This notation is inspired by types in FreshML [34].
- Δ the *bindee parameter environment.* It carries bindings of the form $[x : t_1]t_2$. This environment is needed for type-checking the body of a macro that uses a binder/bindee parameter of the form $\lambda x.x_1$, so that we know that we can only use x_1 at type t_2 in a context where x is bound (with type t_1).
- Π the *late parameter environment.* It carries bindings of the form t. This environment is needed for type-checking the body of a macro that uses a late parameter.
- Γ the *regular environment.* Because we are in a two-stage setting, it carries bindings of the form t^m.

The rules of the MacroML type system are presented in Figure 3. The first three rules deal with variable lookup. The first rule is the variable (projection) rule from MetaML. The next rule is similar, although it reflects the fact that late macro parameters can only be used at level one. The third rule is similar but it checks that the bindee variable is used in a context where its binder variable has already been *bound.*

The next three rules are standard for a multi-level language. All the usual rules of SML would be lifted in the same manner (although some care is required with effects. See Section 7.)

The next four rules are specific to macros. The first rule is for a macro definition. Because we allow macro definitions to be recursive, the body of the macro declaration is checked under the assumption that the macro being defined is already declared. We also add the appropriate assumptions about the binder/bindee parameters, the late parameter, and the early parameter to the appropriate environments. The rest of the rule is standard. Macro application is also analogous to application, although one should note that e_1 and e_2 are checked at different levels. The rules for brackets and escape are special cases of the same rules in MetaML.

4.2 The Semantics of Core MacroML

In this section, we present the definition of the semantics of Core MacroML via an interpretation into MetaML. For any well-typed Core MetaML program $[]; []; []; [] \vdash^1 e : t$ the interpretation $[\![[]; []; []; [] \vdash^1 e : t]\!]$ will define a MetaML term (call it e_0'). To get the final result of running the MacroML program, we simply evaluate the MetaML term $\mathsf{run}\ \langle e_0' \rangle$. To get a more fine-grained view of the evaluation of e_0', we can view it as proceeding into two distinct steps:

- Macro expansion: the MacroML program e expands to a MetaML program e_1' if:

  $$\langle e_0' \rangle \overset{0}{\hookrightarrow} e_1'$$

  where e_0' is the interpretation of the MacroML program into MetaML described above (and defined below).

- Regular execution: The expansion e_1' of e then evaluates to the answer e_2' if:

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

Note that the only new part in the above semantics is the translation from MacroML to MetaML. The two stages of MetaML evaluation are then just standard MetaML rebuilding and evaluation, respectively. Whenever the original term does not have any code types in its MacroML type, the latter step would coincide with standard ML evaluation.

Figure 4 presents the translation, first defined on environments, and then defined on judgments. Although the translation can be made to map untyped terms to untyped terms, it is context-sensitive, and it is therefore easier to define it on judgments of well-typed Core MacroML programs. To avoid the risk of potentially confusing notation, the translation maps judgments to terms (rather than judgments to judgments), as the full MetaML judgments are easy to reconstruct.

Empty environments are mapped to empty environments. The binding for a macro is translated into a MetaML type that, in essence, reflects the semantics of the special notation we have used; the (level 0) MetaML type:

$$t_1 \to \langle t_2 \rangle \to (\langle t_3 \rangle \to \langle t_4 \rangle) \to \langle t_5 \rangle$$

corresponds to a function that takes three (curried) parameters. The first parameter is a "true" value of type t_1 corresponding to the early parameter. The second parameter, however, is a "delayed" or code value of type t_2 corresponding to the late parameter. The third parameter (the binder/bindee parameter) is in fact translated to a function. It is at this point that we can start to explain how the interpretation of the binder/bindee parameters works. Recall that in the examples section we promised to explain how "variables are passed around." The answer is, in fact, that variables are never passed around! During macro-expansion time, the binder/bindee parameter is passed in what can be considered its HOAS representation. The type of a binder/bindee parameter is also translated to a function type. Naturally, this is consistent with the external type of this parameter. The type of a late parameter is simply a "delayed" or code version of the MacroML type of that parameter. No translation is shown for regular environments, as the translation is simply the identity embedding.

The translation on judgments proceeds as follows. Terms that do not involve macros are translated homomorphically. Late parameters are translated by putting an escape around them. The intuition here is that late parameters only occur inside the definition of a macro, and when they occur, they are supposed to be "spliced into" the context where there are used in order to appear in the output of macro expansion as expected.

The key rule in the translations is the one for the bindee parameters: when a bindee parameter x_2 is used in the body of a macro definition, its translation corresponds to an *application* of x_2 to a piece of code carrying the corresponding binder parameter x_1! All of this is escaped so that the application is performed at macro expansion time. To understand what is going on here, keep in mind the translation of the environment Δ, and note that it introduces an arrow type "out of nowhere" during the translation. Thus, in the target of the interpretation, x_2 has an arrow type. This is because, as we said earlier, the translation produces code that is passing around a HOAS representation of the binder/bindee pair.

Macro declarations are translated to escaped function declarations, *i.e.*, function declarations that are to be executed during expansion. Note however that the body of the function being defined and the context where it is used are both in brackets. This is because we want to treat both as code, except in places where the translation has added other escapes.

A macro application is translated into an escaped application, for similar reasons. Note that the first (early) argument to the application is not bracketed, but the second (late) argument is. As hinted earlier by the translation of the types, the binder/bindee argument is translated into a function whose body is itself a piece of code. Intuitively, making the body a piece of code delays its evaluation. It is worth noting that the HOAS entities only exist during the first stage (macro expansion), and not during the execution of the program proper. The translation of brackets and escape is straightforward.

We illustrate the translation on the example at the beginning of the section, with a bound in Γ to the type nat^1:

$$\begin{array}{l}
[\![[];[]; []; a : \mathsf{nat}^1 \vdash^1 \\
\quad \mathsf{letmac}\ f(_, _, \lambda x.y) \ = \lambda a.\lambda x.a + x + y \\
\quad \mathsf{in}\ f((), (), \lambda z.z + a)]\!] = \\
\tilde{}(\mathsf{letrec}\ f\ _\ _\ y = \langle \lambda a.\lambda x.a + x + \tilde{}(y\ \langle x \rangle)\rangle \\
\ \ \mathsf{in}\ \langle \tilde{}(f\ ()\ ()\ (\lambda z.\langle \tilde{}z + a\rangle))\rangle)
\end{array}$$

Environments

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

Lambda Terms

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

$$\frac{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 : t_2 \to t]\!] = e_1' \quad [\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_2 : t_2]\!] = e_2'}{[\![\Sigma;\Delta;\Pi;\Gamma \vdash^m e_1 e_2 : t]\!] = e_1' e_2'}$$

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

Macros

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

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

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

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: Translating MacroML to MetaML

As we said at the start of this section, the output of the translation is wrapped in the context run $\langle \ldots \rangle$ and then evaluated in MetaML to produce the result of macro expansion. Because big-step semantics derivations are hard to draw (and maybe to read), we can "animate" the computation using the following sequence of sound MetaML left-reductions [38].

$\mathsf{run}\ \langle \tilde{}(\mathsf{letrec}\ f\ \_\ \_\ y = \langle \lambda a.\lambda x.a + x + \tilde{}(y\ \langle x\rangle)\rangle$
$\quad \mathsf{in}\ \langle \tilde{}(f\ ()\ ()\ (\lambda z.\langle \tilde{}z + a\rangle))\rangle)\rangle$
$\to$ (call f, rename a when doing substitution)
$\mathsf{run}\ \langle \tilde{}\langle \tilde{}\langle \lambda a'.\lambda x.a' + x + \tilde{}((\lambda z.\langle \tilde{}z + a\rangle)\ \langle x\rangle)\rangle\rangle\rangle$
$\to$ (cancel escape with brackets)
$\mathsf{run}\ \langle \lambda a'.\lambda x.a' + x + \tilde{}((\lambda z.\langle \tilde{}z + a\rangle)\ \langle x\rangle)\rangle$
$\to$ (call inner λ)
$\mathsf{run}\ \langle \lambda a'.\lambda x.a' + x + \tilde{}\langle \tilde{}\langle x\rangle + a\rangle\rangle$
$\to$ (cancel escape with brackets)
$\mathsf{run}\ \langle \lambda a'.\lambda x.a' + x + (x + a)\rangle$
$\to$ (run)
$\lambda a'.\lambda x.a' + x + (x + a)$

which is the expected result of macro expansion.

4.3 Type Safety

As mentioned in the introduction, defining the semantics of Core MacroML by interpretation into MetaML makes proving type safety fairly direct. In what follows, we state and outline the proof of this result.

THEOREM 4.1 (TYPE SAFETY). *If* $[];[];[];[] \vdash^m e : t$ *is a valid MacroML judgment, then translating it to MetaML yields a well-typed MetaML program, and executing that program does not generate any MetaML runtime errors.*

PROOF. The first part is by Lemma 4.2, and the second part follows from the type safety property of the MetaML type system presented in previous work [40, 26]. □

In the statement of the theorem, MetaML runtime errors include both errors that might occur at macro expansion and run-time errors (defined precisely in [40, 26, 38]). The necessary auxiliary lemma states that our translation preserves typing.

LEMMA 4.2 (TYPE PRESERVATION). *If* $\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t$ *is a valid MacroML judgment, then* $[\![\Sigma]\!], [\![\Delta]\!], [\![\Pi]\!], \Gamma \vdash^m [\![\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t]\!] : t$ *is a valid MetaML judgment.*

PROOF. Routine induction on the height of the derivation of $\Sigma; \Delta; \Pi; \Gamma \vdash^m e : t$. □

5. PRACTICAL EXTENSIONS OF CORE

Core MacroML handles simple functional inlining, recursive inlining, and the definition of simple binding constructs. By design, Core MacroML is a minimalistic language whose purpose is to demonstrate how the main semantic subtleties of a typed macro system can be addressed. We have implemented Core MacroML using a toy implementation of MetaML. We have used the implementation to run a benchmark of simple programs in Core MacroML, and the results have consistently been as expected. In this section, we describe extensions of Core MacroML with additional features that would be desirable in a practical implementation. We expect that all these extensions are not hard to implement.

Let Bindings: In the introduction, we have presented examples of `let`-expression macros with only one binding. However, the same macro definition can be used to expand `let`-expressions with multiple bindings. For this purpose, we propose the use of a comprehension-like notation to allow the user to express such macros. For example, the expression:

```
let mac (let seq x{i} = e1{i} in e2 end) =
       $(let val x{i} = (print (Int.toString i);
                         ?e1{i})
         in ?e2 end)
in let seq y = f $7
       seq z = h y
   in g z end
end
```

would expand to:

```
$(let val x1 = (print (Int.toString 1); ?(f $7))
      val x2 = (print (Int.toString 2); ?(h x1))
  in ?(g x2) end)
```

where it becomes apparent that `i` is an implicit comprehension parameter that gets bound to the index of the binding under consideration, and that `x{i}` and `e{i}`, are the parameters for this i^{th}-binding.

Note that the number of declarations (the range of `i`) will be known at translation time, as it is manifest from the application of the macro. However, because vanilla MetaML does not have support for constructing declarations of arbitrary length, the most direct approach to interpret this proposal would be to produce one MetaML function for each macro application. This trick is similar to polyvariant specialization in partial evaluation [18]. The obvious disadvantage of this approach is that it inflates the size of MetaML. We would like to explore extensions to MetaML that would allow us to interpret this new construct in a more natural fashion.

Recursive Bindings: A simple but still important issue is how recursive binding constructs should be treated. In particular, when a macro is defining a new binding construct in terms of an existing recursive binding construct, this information should be maintained in the type of the macro. Consider the following declaration:

```
let mac (let fin x{i} _ = e1{i} in e2 end)
      = (let fun x{i} _ = e1{i} in e2 end)
in ... end
```

This declaration may appear ambiguous because we can *either* expand the `fun` comprehension into a sequence of `fun` declarations or a sequence of mutually-recursive ("`and`ed") `fun` declarations. However, this can be completely determined by how the `fin` construct is used: if it is used as a disjoint sequence, then that is what should be produced. If it is used as an `and`ed sequence, then the result should be like-wise. In the latter case, however, we need to check that the parameters to the `and`ed sequences of `fin`s should not have duplicate variables names. All these checks can be done statically.

Dist-fix Operators: Finally, we come to an extension that is rather orthogonal to the rest of our proposal. However, in practical macro systems, it is a valuable addition. In particular, it is relatively easy to add dist-fix operators to our language. The key idea is that each macro definition should still be determined by the first symbol used in its name. With such an extension, it is possible to define some other basic constructs in a language:

```
let mac {if, then, else} if c then t else f =
        case c of
          true  => t
        | false => f
in if true then 1 else 2 end
```

The syntax simply extends what we have seen before by a declaration of keywords that can be used in conjunction with the macro `if`. Then, the rest of the argument list dictates the "dist-fix arity" of this macro. The only complication with the introduction of such macros is that they make parsing context sensitive. However, this is already the case in SML because of infix operators.

6. RELATED WORK

Our approach for deriving the type system for MacroML was to first develop the translation in an essentially untyped setting, and then to develop a type system that characterizes when the result of the translation is a well-typed MetaML program. The earliest instance of such a translation appears in a work by Wand [43]. Because we start with the untyped setting, we expect that similar derivations are possible for richer MetaML type systems (including features such as polymorphism and effects, for example).

The earliest use of a binary type constructor to indicate the type of a "piece of code with a free variable in it" such as our $[t_1]t_2$ seems to have been in Miller's proposal for "an extension of ML to handle bound variables in data structures." Miller's proposal is more ambitious than ours in that it tries to deal with data types that have some binding structure, but it neither addresses the issue of defining new binding constructs in a user-level language nor gives a formal semantics for the proposed constructs. Indeed, work by Pašalić, Sheard and Taha suggests that Miller's proposal may need to be reformed before it can have a simple semantics [32]. More recently, FreshML has also used a similar binary type constructor based on a denotational model [34]. The Twelf system uses a mixture of dependent types that seems to be, at least intuitively, similar to our $[x : t_1]t_2$ construction. To our knowledge, our work seems to be the first to investigate the application of such type systems directly to the domain of macro systems, and to expose the connections with multi-stage languages and higher-order syntax.

Finally, the title of the paper by Bawden [3] suggests that it is related to the present work, but it does not seem to address the issue of typing or type systems formally.

7. CONCLUSION

We have presented a proposal for a typed macro system, and have shown how it can be given a rigorous yet readable semantics via an interpretation into the multi-stage programming language MetaML. The interpretation is essentially a denotational semantics where MetaML is the internal language of the model. Such models have already been studied elsewhere [4]. But because MetaML enjoys a simple and intuitive operational semantics, our proposal is easy to implement in a directly usable form.

The macro language that we have presented, MacroML, has useful and novel features, combining both static typing and allowing the user to define new binding constructs. In trying to achieve this, we have used ideas from both HOAS (to implement our proposal in a multi-stage setting) [33] and FreshML (to provide the surface syntax and ideas in the source language) [34]. It may well be that our language provides some new insights on the link between the two approaches to treating binding constructs.

We have argued that macros are useful. But the moral of the paper is of a more technical nature: *multi-stage programming languages are a good foundation for the semantics-based design of macro systems.* We have shown how a formal multi-stage interpretation of macro systems provides an elegant way of avoiding binding issues, defining new binding constructs, and provides a sound basis for developing type systems for macro languages.

In this paper, we have not considered type safety in the presence of imperative features (references, exceptions) during expansion time. In this setting, we expect the work on imperative multi-level languages to be of direct relevance [42, 5]. We have also not considered a multi-level macro system primarily for the reason of simplicity. We would like to consider such an extension in future work. But there are restrictions on the system that may be a bit more challenging to alleviate. For example, we have not considered higher-order macros (macros that take other macros as parameters) and we have not considered macros that generate other macros. For such expressiveness, however, we expect that it may be simpler and more appropriate to move directly to a full-fledged multi-stage programming language. Part of the appeal of macro systems, we believe, goes away when we attempt to push them to the higher-order and reflective setting. On the other hand, given that we have defined macros in terms of a multi-stage language, it should be possible to merge macros and MetaML into the same language without any surprising interactions.

To conclude, while this paper addresses key semantic concerns in developing an expressive, type-safe macro system, this is only a start. We have only built a simple prototype during this work. The prototype involved a direct implementation of MetaML semantics and a direct implementation of the translation. In the future, we hope to integrate this work with ongoing work on multi-stage extensions of the major statically-typed functional languages, namely SML, Ocaml, and Haskell.

8. REFERENCES

[1] Ariola, Z. M., and Sabry, A. Correctness of monadic state: An imperative call-by-need calculus. In *ACM Symposium on Principles of Programming Languages* (1998), ACM Press, pp. 62–74.

[2] 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.

[3] Bawden, A. First-class macros have types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)* (N.Y., Jan. 19–21 2000), ACM Press, pp. 133–141.

[4] 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).

[5] Calcagno, C., Moggi, E., and Taha, W. Closed types as a simple approach to safe imperative multi-stage programming. In *27th International Colloquium on Automata, Languages, and Programming (ICALP)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, ACM Press, pp. 25–36.

[6] Curry, H. B., and Feys, R. *Combinatory Logic, Volume I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. Second printing 1968.

[7] Davies, R. A temporal-logic approach to binding-time analysis. In *11^{th} Annual IEEE Symposium on Logic in Computer Science (LICS)* (New Brunswick, 1996), IEEE Computer Society Press, pp. 184–195.

[8] Davies, R., and Pfenning, F. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg Beach, 1996), pp. 258–270.

[9] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (Dec. 1992), 295–326.

[10] Erkök, L., and Launchbury, J. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00* (September 2000), ACM Press, pp. 174–185.

[11] Felleisen, M. On the expressive power of programming languages. In *Science of Computer Programming* (1991), vol. 17, pp. 35–75. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

[12] Fiore, M., Plotkin, G., and Turi, D. Abstract syntax and variable binding. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)* (Trento, Italy, July 1999), G. Longo, Ed., IEEE Computer Society Press, pp. 193–202.

[13] Glück, R., and Jørgensen, J. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs (PLILP'95)* (1995), S. D. Swierstra and M. Hermenegildo, Eds., vol. 982 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 259–278.

[14] Glück, R., and Jørgensen, J. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics* (1996), D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 261–272.

[15] Glück, R., and Jørgensen, J. An automatic program generator for multi-level specialization. *LISP and Symbolic Computation 10*, 2 (1997), 113–158.

[16] Gomard, C. K., and Jones, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming 1*, 1 (1991), 21–69.

[17] Hofmann, M. Semantical analysis of higher-order abstract syntax. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)* (Trento, Italy, July 1999), G. Longo, Ed., IEEE Computer Society Press.

[18] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[19] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. In *Proceedings of the ACM Conference on LISP and Functional Programming* (Cambridge, MA, Aug. 1986), R. P. Gabriel, Ed., ACM Press, pp. 151–181.

[20] Lewis, J. R., Launchbury, J., Meijer, E., and Shields, M. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)* (N.Y., Jan. 19–21 2000), ACM Press, pp. 108–118.

[21] Miller, D. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop* (June 1990). Available as UPenn CIS technical report MS-CIS-90-59.

[22] Mitchell, J. C. On abstraction and the expressive power of programming languages. In *Theoretical Aspects of Computer Software* (1991), T. Ito and A. R. Meyer, Eds., vol. 526 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 290–310.

[23] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991).

[24] Moggi, E. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics* (1997), Elsevier Science.

[25] 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.

[26] 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.

[27] Moran, A., and Sands, D. Improvement in a lazy context: An operational theory for call-by-need. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas* (New York, NY, Jan. 1999), ACM, pp. 43–56.

[28] Nielson, F., and Nielson, H. R. Two-level semantics and code generation. *Theoretical Computer Science 56*, 1 (1988), 59–133.

[29] Okasaki, C. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[30] Oliva, D. P., Ramsdell, J. D., and Wand, M. The VLISP verified prescheme compiler. *Lisp and Symbolic Computation 8*, 1/2 (1995), 111–182.

[31] 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.

[32] Pašalić, E., Sheard, T., and Taha, W. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Tech. Rep. CSE-00-007, OGI, 2000. Available from [31].

[33] Pfenning, F., and Elliott, C. Higher-order abstract syntax. In *Proceedings of the Symposium on Language Design and Implementation* (Atlanta, 1988), pp. 199–208.

[34] Pitts, A. M., and Gabbay, M. J. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Programme Construction* (2000), vol. 1837 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–255.

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

[36] Sabry, A. What is a purely functional language? *Journal of Functional Programming 8*, 1 (Jan. 1998), 1–22.

[37] Sheard, T. Using MetaML: A staged programming language. *Lecture Notes in Computer Science 1608* (1999), 207–??

[38] Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [31].

[39] 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.

[40] 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 (ICALP)* (Aalborg, 1998), vol. 1443 of *Lecture Notes in Computer Science*, pp. 918–929.

[41] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)* (Amsterdam, 1997), ACM Press, pp. 203–217.

[42] Thiemann, P., and Dussart, D. Partial evaluation for higher-order languages with state. Available online from http://www.informatik.uni-freiburg.de/~thiemann/papers/index.html, 1996.

[43] Wand, M. Embedding type structure in semantics. In *Conference Record of the Principles of Programming Languages* (1985), pp. 1–6.

[44] Wand, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation 10* (1998), 189–199.