Environment Classifiers

Extended Abstract

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

Michael Florentin Nielsen†
Department of Computer Science
IT University of Copenhagen
erik@it-c.dk

ABSTRACT

This paper proposes and develops the basic theory for a new approach to typing multi-stage languages based a notion of *environment classifiers*. This approach involves explicit but lightweight tracking – at type-checking time – of the origination environment for future-stage computations. Classification is less restrictive than the previously proposed notions of closedness, and allows for both a more expressive typing of the "run" construct and for a unifying account of typed multi-stage programming.

The proposed approach to typing requires making cross-stage persistence (CSP) explicit in the language. At the same time, it offers concrete new insights into the notion of levels and in turn into CSP itself. Type safety is established in the simply-typed setting. As a first step toward introducing classifiers to the Hindley-Milner setting, we propose an approach to integrating the two, and prove type preservation in this setting.

Categories and Subject Descriptors

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

General Terms

Design, Reliability, Languages, Theory, Verification.

Keywords

Multi-stage programming, type systems, type safety, linear temporal logic, modal logic.

*Supported by NSF ITR-0113569. This work was done at Yale University.
†Work done while visiting Yale University

1. INTRODUCTION

At the untyped level, multi-stage languages are a unifying framework for the essence of partial evaluation [32, 11, 10, 34, 25, 52], program generation [36, 59, 3], runtime code generation [2, 17, 18, 28, 29], and generative macro-systems [21]. But until now, this was not the case for the typed setting, as there have been different, orthogonal proposals for typing multi-stage languages. In fact, ever since the inception versions of these formalisms, including those of Nielson and Nielson [44, 46, 47, 51, 48, 45, 49, 50] on one hand, and Gomard and Jones [35, 26, 27, 34] on the other, the question of whether these languages should support either closed code or open code has been unresolved. Code is *open* if fully evaluated runtime value of this type can contain free variables. Code is closed otherwise. Note that traditionally in both call-by-name and call-by-value languages, values are closed. Languages supporting closed code naturally allow for *safe* runtime execution of this code. Languages supporting open code naturally admit a form of symbolic computation. Combining the two notions has been the goal of numerous previous works.

1.1 Multi-stage Basics

Multi-stage programming languages provide a small set of constructs for the construction, combination, and execution of delayed computations. Programming in a multi-stage language such as MetaOCaml [39] can be illustrated with the following classic example[1]:

```
let even n = (n mod 2) = 0
let square x = x * x
let rec power n x = (* int -> .<int>. -> .<int>. *)
  if n=0 then .<1>.
   else if even n
         then .<square .~(power (n/2) x)>.
         else .<.~x * .~(power (n-1) x)>.
let power72 =                     (* int -> int *)
  run .<fun x -> .~(power 72 .<x>.)>.
```

Ignoring the type constructor `.<`*t*`>.` and the three staging annotations brackets `.<`*e*`>.`, escapes `.~`*e* and `run`, the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^{72}. Without the staging, however, the last step just produces a closure that invokes the power function every time

[1]Dots are used around brackets and escapes to disambiguate the syntax in the implementation. They are dropped when we talk about underlying calculus rather than the implementation.

it gets a value for x. To understand the effect of the staging annotations, it is best to start from the end of the example. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application e `.<x>.` has to be performed even though x is still an uninstantiated *symbol*. In the `power` example, `power 72 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the `power` function, the recursive applications of `power` are also escaped to make sure that they are performed immediately. The `run` on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.

1.2 When is Code Executable?

A basic challenge in designing typed multi-stage languages is ensuring that future stage code can be executed in a type-safe manner. In particular, not all code arising during a multi-stage computation is executable. In the expression such as

`.<fun x -> .~(`e `.<x>.)>.`

the sub-expression e should not attempt to execute the term `.<x>.`. For example, e should not be `run`.

1.2.1 State of the Art

Moggi pointed out that early approaches to multi-level languages seem to treat code either as always being open or always being closed [40]. These two approaches are best exemplified by two type systems motivated by different systems from modal logic:

$\lambda^{\bigcirc}$ Motivated by the $\bigcirc$ modality from linear time temporal logic, this system provides a sound framework for typing constructs that have the same operational semantics as the bracket and escape constructs mentioned above [14]. As illustrated above, the brackets and escapes can be used in to annotate λ-abstractions so as to force symbolic computations. This type system supports generating code, but does not provide a construct corresponding to `run`.

$\lambda^{\Box}$ Motivated by the $\Box$ modality from modal logic, this system provides constructs for constructing and running code [15]. Until now, the exact correspondence between these constructs and brackets, escape, and `run` where not established. It is known, however, that all values generated during the evaluation of terms in this language are closed. Thus, the connection between this system and symbolic evaluation is less obvious than for the previous system.

There have been two distinct efforts to combine the features of these two type systems:

λ^{R} The essence of this approach is to count the number of `run`'s surrounding the term `.<x>.` [42]. Unfortunately, this means that `run` cannot be lambda-abstracted. This is a serious expressivity limitation, because we cannot type

`let x = .<1+1>. in run x.`

when let is seen as syntactic sugar for a beta-redex. The "`run` counting" approach propagates the information about the presence of `run` and its impact on the value it takes as its argument by change only the *locally perceived typing of the environment.*

λ^{BN} This approach uses combinations of the temporal $\bigcirc$ modality and the modal logic $\Box$ modality to allow both open code and a means to express that a certain code fragment is closed [42, 5, 8, 6, 7]. Using a type constructor for *closed values* it is possible to assign `fun x -> run x` the type `[.<A>.]->[A]`, which means that `run` takes a closed code fragment and returns a closed value. But not all open code is unsafe to run. Additionally, this approach is acceptable for executing single-stage programs (which do not contain future stage variables), but it is unnatural in the multi-stage setting, and treats open code as a second class citizen.

Thus, while the first approach allows us to run open code but not abstract `run`, the second allows us to do the latter but not the former. This paper presents a system that provides both features.

1.3 Implicit Goals

The explicit goal of the present work is to address the above limitations. This can be viewed as a quest for more accurate types for multi-stage program. But an implicit goal is to improve (or at least maintain) the particularly light-weight nature of previous proposals for syntax, types, and equational theory for multi-stage languages. This goal explains why explicit representations of future stage code are not solutions that we will consider (c.f [60]). For example, representing future stage computations by strings or even datatypes does not provide the programmer with any help in avoiding the inadvertent construction of syntactically incorrect or ill-typed programs. A better alternative would be the use of dependently typed inductive datatypes (as is done for object languages programs in the recent work on Meta-D [54]). This option, however, is substantially more verbose than using a quotation mechanism like brackets and escape. In addition, it forces the user to move to the dependently typed setting and to give up Hindley-Milner type inference. More importantly, it does not provide any immediate support for avoiding inadvertent variable capture problems, unless we also have access to FreshML types [55, 43], which have not yet been studied in the dependent type setting. Finally, all such representations are *intensional.* While this allows the programmer to express program transformations, it reduces the equational theory on future stage computations to syntactic equivalence [62, 67]. This reduction is not always desirable.

1.4 Approach

We have considered reflecting the free variables of a term explicitly in the types of open fragments, and borrowing ideas from linking calculi [9], implicit parameters [38], module systems [57], explicit substitutions [4], and program analysis [19]. The general flavor of such an approach can be illustrated by the following hypothetical judgment:

$$\Gamma \vdash \langle x \rangle : \langle t \rangle^{\{x:t\}}$$

Closer inspection of this approach suggests that it may not be as appealing as it seems at first. A practical programming concern with this approach is that types become very big, as the size of a type would be linear in the number of variables used in the term. Furthermore, to deal with α-renaming correctly, variable uses should not be directly mentioned in types. Thus, the term language needs to be extended to handle this problem. In addition, a notion of polymorphism would be essential. For example, the term

```
fun f -> .<fun x -> .~(f .<x>.)>.
```

is a two-level η-expansion [13], and is used quite often in multi-stage programming. It would be highly desirable to have a single type for this function, so that it would not be defined afresh every time we need it. This function would need to take a value of type $\langle t_1\rangle^e \to \langle t_2\rangle^e$ and to return a value of type $\langle t_1 \to t_2\rangle^e$ for "any" environment e. A reasonable approach to dealing with this problem is to introduce ρ polymorphism [66], but we would need to use negative side conditions in the type system to stop the type $\{x : t\}$ from going outside the scope of this term. Negative side conditions complicate unification, and in turn inference. It is also unclear how this approach could be generalized to the multi-level setting.

1.5 Organization and Contributions

Section 2 explains the need for explicit cross-stage persistence (CSP), which is necessary for the development of the notion of environment classifiers. The key insight behind the work presented in this paper is that *a carefully crafted notion of polymorphic variables that are never concretely instantiated is both possible and sufficient for providing an expressive type system for multi-stage programming* and at the same time avoiding the typing problems associated with introducing signatures into types. These variables will be called environment classifiers, and are introduced in Section 3.

The soundness of Hindley-Milner polymorphism in the presence of statically typed brackets and escapes does not seem to have been addressed in the literature (c.f. [7]). Dynamically typed versions of brackets, escape, and `run` have been used to introduce a notion of staged type inference [58]. Type safety has been demonstrated in this setting, but it was also found that care must be taken in defining the semantics of such language. Section 4 demonstrates the soundness of typed multi-stage programming in the presence of Hindley-Milner polymorphism.

Section 5 shows how a type system based on this approach provides a unifying account of typed multi-stage programming. This is demonstrated by presenting two embeddings of both $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ into the new typed language. The new type system has no more constructs than the previous systems.

2. CROSS-STAGE PERSISTENCE

The proposed approach requires the use of an explicit treatment of cross-stage persistence. At the same time, it sheds a new light on this basic notion. *Cross-stage persistence (CSP)* [64] is the ability to use a value available in the current stage in a future stage. If a user defined the list `reverse` function, there is usually no reason to prohibit them from using `reverse` inside future-stage code. This feature is used to allow the usage of both `*` and `square` inside brackets in the `power` example.

The treatment of this feature has subtle interactions with the `run` typing problem. Type theoretically, CSP can be introduced in two distinct ways: Implicit and explicit. With the implicit approach (used in the `power` example above), CSP is only admitted on variables through a rule such as:

$$\frac{\Gamma(x) = t^m \quad m \le n}{\Gamma \overset{n}{\vdash} x : t}$$

which compares the bracket-depth at which a variable is bound (m) and makes sure that it is no greater than that at which the variable is used (n). It must then be separately demonstrated the there is an analogous notion of implicit CSP (called "promotion") that holds on terms [63, 42]. The implicit approach, therefore, allows less structure than what we show here to be possible and is essential for executing open terms. In particular, the implicit approach (and developments based on it [7]) considers a term to be cross-stage persistent if and only if its variables can be made cross-stage persistent. With the explicit approach, there is a dedicated construct with the following typing:

$$\frac{\Gamma \overset{n}{\vdash} e : t}{\Gamma \overset{n+1}{\vdash} \%e : t}$$

which essentially allows us to pretend that we are surrounded by one less bracket when we are typing the term e. To illustrate the kind of structure missing in the implicit presentation, consider the term:

```
.<fun x -> .~(run .<.<x>.>.)>.
```

which evaluates under the standard untyped big-step semantics for multi-stage languages (c.f. [62]) to `.<fun x -> x>.`. In the initial term, `x` is surrounded by only one bracket when it is bound, and two brackets when it is used. Implicit CSP means that we can consider the argument to `run` to be a term `.<.<...>.>.` containing a cross-stage persistent constant `x`. This is in fact *not* a reasonable interpretation, because the inner brackets will eventually be canceled by the escape in the original term. Thus, the variable `x` is surrounded immediately by the "right" brackets, and should not be treated as a cross-stage persistent constant. An acceptable interpretation, however, is that the argument to `run` is a term `.<...>.` containing a cross-stage persistent constant `.<x>.`. The intuition is that inside the escape `.<x>.` is a legitimate (typable) value, and we wish to use it inside another pair of brackets. Equationally, these observations can be interpreted as:

$$\langle \%\langle e\rangle\rangle \not\approx \langle\langle \%e\rangle\rangle$$

for any notion of typed equality $\approx$ that we would be interested in. *This distinction cannot be made if CSP is implicit.* Exactly why explicit CSP is useful for preserving the semantics of classification becomes clear in details of the demotion lemma. This lemma is the cornerstone in the argument for why running a given code fragment preserves typing.

3. ENVIRONMENT CLASSIFIERS

We introduce the idea of environment classifiers by means of a multi-stage calculus called λ^α. After presenting its type system and semantics, we demonstrate its basic properties, such as type safety. We also relate the various features of λ^α to previous proposals.

$$\begin{array}{lcccl}
\text{Environment Classifiers} & \alpha, \beta & \in & W & \\
\text{Named Levels} & A & \in & W^* & ::= \; \alpha_1, \ldots, \alpha_n \\
\text{Types} & t & \in & T & ::= \; \mathsf{int} \mid t_1 \to t_2 \mid (\alpha)t \mid \langle t \rangle^\alpha \\
\text{Environments} & \Gamma & \in & G & ::= \; [] \mid \Gamma, x : t^A \mid \Gamma, \alpha \\
\text{Expressions} & e & \in & E & ::= \; i \mid x \mid \lambda x.e \mid e_1\, e_2 \mid (\alpha)e \mid e[\alpha] \mid \langle e \rangle^\alpha \mid \,\tilde{}\, e \mid \mathsf{run}\ e \mid \%e
\end{array}$$

Types, Environments, and Named Levels:

$$\frac{\alpha \in dom(\Gamma)}{\Gamma \vdash \alpha} \qquad \frac{GV(t) \subseteq dom(\Gamma)}{\Gamma \vdash t} \qquad \frac{}{\vdash []} \qquad \frac{\vdash \Gamma \quad \alpha \notin \Gamma}{\vdash \Gamma, \alpha} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash t \quad \Gamma \vdash \alpha_i}{\vdash \Gamma, x : t^{\alpha_1, \ldots, \alpha_n}} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \alpha_i}{\Gamma \overset{\alpha_1, \ldots, \alpha_n}{\vdash}}$$

Terms:

$$\frac{\Gamma \overset{A}{\vdash}}{\Gamma \overset{A}{\vdash} i : \mathsf{int}} \qquad \frac{\Gamma(x) = t^A \quad \Gamma \overset{A}{\vdash}}{\Gamma \overset{A}{\vdash} x : t} \qquad \frac{\Gamma, x : t_1^A \overset{A}{\vdash} e : t_2}{\Gamma \overset{A}{\vdash} \lambda x.e : t_1 \to t_2} \qquad \frac{\Gamma \overset{A}{\vdash} e_1 : t_1 \to t_2 \quad \Gamma \overset{A}{\vdash} e_2 : t_1}{\Gamma \overset{A}{\vdash} e_1\, e_2 : t_2}$$

$$\frac{\Gamma, \alpha \overset{A}{\vdash} e : t}{\Gamma \overset{A}{\vdash} (\alpha)e : (\alpha)t} \qquad \frac{\Gamma \overset{A}{\vdash} e : (\alpha)t \quad \Gamma \vdash \beta}{\Gamma \overset{A}{\vdash} e[\beta] : t[\alpha := \beta]} \qquad \frac{\Gamma \overset{A,\alpha}{\vdash} e : t \quad \Gamma \vdash \alpha}{\Gamma \overset{A}{\vdash} \langle e \rangle^\alpha : \langle t \rangle^\alpha} \qquad \frac{\Gamma \overset{A}{\vdash} e : \langle t \rangle^\alpha}{\Gamma \overset{A,\alpha}{\vdash} \,\tilde{}\, e : t} \qquad \frac{\Gamma \overset{A}{\vdash} e : t \quad \Gamma \overset{A,\alpha}{\vdash}}{\Gamma \overset{A,\alpha}{\vdash} \%e : t} \qquad \frac{\Gamma \overset{A}{\vdash} e : (\alpha)\langle t \rangle^\alpha}{\Gamma \overset{A}{\vdash} \mathsf{run}\ e : (\alpha)t}$$

Figure 1: Syntax and Static Semantics of Simply Typed λ^α

NOTATION 1. *We define sets of terms in three ways:*

$::=$ *is followed by a set of productions that can be used to derive a term in the set being defined.*

$\overset{+}{::=}$ *is followed by a set of productions that are allowed in* addition *to ones previously introduced.*

$=$ *is standard set equality.*

3.1 Basic Definitions

Figure 1 presents the static semantics of λ^α. The first syntactic category introduced is *environment classifiers*, ranged over by α and β, drawn from a countably infinite set of names W. Environment classifiers allow us to name parts of the environment in which a term is typed. *Named levels* are sequences of environment classifiers. They will be used to keep track of which environments are being used as we build nested code. Named levels are thus an enrichment of the traditional notion of levels in multi-stage languages [14, 64, 62], the latter being a natural number which keeps track only of the depth of nesting.

The first two type constructs are standard: Integers int, and functions $t_1 \to t_2$. Next is α-closed types $(\alpha)t$, read "α-closed t".[2] Note that the occurrence of α here is in a binding position, and it can occur free in t. Last is the type for code fragments $\langle t \rangle^\alpha$, read "code t in α". Here α is in a usage occurrence. Compared to previous work, the $\langle t \rangle^\alpha$ type is a refinement of $\lambda^{\bigcirc}$'s $\bigcirc t$ [14] (the latter also being essentially the open code type used in previous proposals for type systems for MetaML [63, 42, 60].) The α in this type is an abstract reference to the environment in which this code fragment was created.

Environments are ordered sequences that can carry either a variable declaration $x : t^A$ indicating that x is a variable of type t that has been introduced at named level A, or an environment declaration α. Additional well-formedness requirements are given, but first we introduce expressions.

The first four kinds of *expressions* are standard: integers i, variables x drawn from some countably infinite set of names, lambda abstractions $\lambda x.e$, and applications $e_1\ e_2$. The next two constructs are α introduction $(\alpha)e$, read "α-closed e", and instantiation $e[\alpha]$, read "e of α". They allow the programmer to declare the start of new, named environments and the embedding of an α-closed value into a weaker context. The next three constructs are essentially the standard constructs of MetaML [64]: Brackets $\langle e \rangle^\alpha$, escape $\tilde{}\,e$, and run run e. Having an α annotation, however, is not standard. While this annotation has no effect on the operational semantics, it is a declaration of the environment this code fragment should be associated with. We take the last α in the environment as the default, and require the user to provide one explicitly only when it is different from the default. Next is an explicit construct for CSP $\%e$. In MetaML, CSP is limited (by typing rules) to variables [64]. In the current proposal, however, CSP will be allowed on terms.

As mentioned before, environments are ordered. There are additional requirements on environments. A type is well-formed under a given environment, written $\Gamma \vdash t$, when all the free environment classifiers in t, written $GV(t)$, are contained in the declarations in Γ. An environment is well-formed, written $\vdash \Gamma$, when all free variables in a type and the named level for a binding are introduced in the environment to its left.

3.2 Type System

The first four rules are mostly standard. As in type systems for multi-level languages, the named level A is propagated without alteration to the sub-terms in these constructs. In the variable rule, the named level associated with the variable being typed is checked and is required to be the *same* as the current level. In the lambda abstraction rule, the named level of the abstraction is recorded in the environment.

[2]This quantifier is similar to universal quantification in system F. The main difference is that it is restricted to a particular kind, and never instantiated to a concrete value. As the full implications of these restriction are not yet known, we have avoided using the notation $\forall$ for this quantifier.

$$\begin{array}{lcccl}
\text{Levels} & n, m & \in & N & ::= & 0 \mid n+ \\
\text{Stratified Expressions} & e^n & \in & E^n & ::= & i \mid x \mid \lambda x.e^n \mid e^n\ e^n \mid \langle e^{n+} \rangle^\alpha \mid \mathsf{run}\ e^n \mid\ (\alpha)e^n \mid\ e^n[\alpha] \\
& e^{n+} & \in & E^{n+} & ::\overset{+}{=} & \tilde{}\,e^n \mid\ \%e^n \qquad \text{(Note definition of } E^n \text{ above also).} \\
\text{Values} & v^0 & \in & V^0 & ::= & i \mid \lambda x.e^0 \mid \langle v^1 \rangle^\alpha \\
& v^{n+} & \in & V^{n+} & := & e^n \mid \%v^n
\end{array}$$

Demotion (and Auxiliary Definitions):

$$i\downarrow_A^X \equiv i, \quad x\downarrow_A^X \equiv x, \quad \lambda x.e\downarrow_A^X \equiv \lambda x.(e\downarrow_A^{X,x:A}), \quad e_1\ e_2\downarrow_A^X \equiv e_1\downarrow_A^X\ e_2\downarrow_A^X, \quad (\alpha)e\downarrow_A^X \equiv (\alpha)(e\downarrow_A^X), \quad e[\alpha]\downarrow_A^X \equiv e\downarrow_A^X [\alpha],$$

$$\langle e \rangle^\alpha \downarrow_A^X \equiv \langle e\downarrow_{A,\alpha}^X \rangle^\alpha, \quad \tilde{}\,e\downarrow_{A,\alpha}^X \equiv \tilde{}\,(e\downarrow_A^X), \quad \mathsf{run}\ e\downarrow_A^X \equiv \mathsf{run}\ (e\downarrow_A^X), \quad \%e\downarrow_{\alpha,A,\alpha'}^X \equiv \%(e\downarrow_{\alpha,A}^X), \quad \%e\downarrow_\alpha^{x_i:A_i} \equiv e[x_i := \%_{A_i} x_i]$$

$$\%_\alpha e \equiv \%e, \quad \%_{\alpha,A,\alpha'} e \equiv \tilde{}\,(\%_{\alpha,A}\langle e \rangle^{\alpha'}), \quad \%(\Gamma,\alpha) \equiv \%\Gamma, \quad \%(\Gamma, x : t^A) \equiv (\%\Gamma, x := \%_A x)$$

$$|(\Gamma,\alpha)| \equiv |\Gamma|, \quad |(\Gamma, x : t^A)| \equiv (|\Gamma|, x : A), \quad (\Gamma,\alpha)^A \equiv \Gamma^A, \alpha, \quad (\Gamma, x : t^{A'})^A \equiv (\Gamma, x : t^{A,A'})$$

Notions of Reduction:

$$\begin{array}{lcl@{\qquad\qquad}rcl}
(\lambda x.e^0)\, v^0 & \longrightarrow_\beta & e^0[x := v^0] & \tilde{}\,\langle v^1 \rangle^\alpha & \longrightarrow_E & v^1 \\
((\alpha)v^0)[\beta] & \longrightarrow_W & v^0[\alpha := \beta] & \mathsf{run}\ (\alpha)\langle v^1 \rangle^\alpha & \longrightarrow_R & (\alpha)(v^1 \downarrow_\alpha^{[]})
\end{array}$$

Figure 2: Stratified Expressions, Values, Demotion, and Reductions for λ^α

The next two rules are for α introduction and instantiation. For a (new) α to be introduced around a type, the term of that type must be typable under the current environment extended with α at the top. The type system and the well-formedness conditions ensure that α was not introduced previously by the environment Γ. The well-formedness conditions on environments therefore ensure that α can only occur in environments used higher up in the type derivation tree, and to the right of this α. Proof-theoretically, these environments are the ones that α classifies. The rule for α instantiation allows replacing α in any α-closed type by any classifier β in the current environment.

The rule for brackets is almost the same as in $\lambda^\bigcirc$ and in previous type systems for MetaML. First, for every code type a classifier must be assigned. This classifier must already be declared in the environment. Second, while typing the body of the code fragment inside brackets, the named level of the typing judgment is extended by the name of the "current" classifier. This information will be used in both the variable rule and the escape rule to make sure that only variables and code fragments of the same classification are ever incorporated into this code fragment. The escape rule at level A, α only allows the incorporation of code fragments of type $\langle t \rangle^\alpha$.

The rule for CSP itself is standard: It allows us to incorporate a term e at a "higher" level. However, we will see in the rest of the section that using named levels has a significant effect on the role of CSP. Finally, the rule for run allows us to execute a code fragment of type $\langle t \rangle^\alpha$ only if it is α-closed.

3.3 Reduction Semantics

Figure 2 presents the definition of the reduction semantics for λ^α, and some auxiliary definitions needed for characterizing the behavior of typings under reductions.

Levels are natural numbers. *Stratified expressions* are used to define *values*. There are two essential properties that stratification is used to capture: First, escapes can only occur at levels $n+$ in expressions and $n++$ in values. Second, % can only occur at level $n+$ in both expressions and values.

Demotion is an operation needed to convert a value $\langle v^1 \rangle$ into an expression e^0. This is needed to carry out the run reduction, which takes a term "free of level 1 escapes" and runs it. In multi-stage calculi where there is no explicit CSP, this operation is syntactic identity (c.f.[42]). The definition here is similar to previous definitions used in calculi with explicit CSP (c.f. [7]), except that previous work defines demotion in the case of % at the lowest level (0) as:

$$\%e\downarrow_0 \equiv e[x_i := \%x_i] \quad \{x_i\} \equiv FV(e)$$

The definition presented here:

1. uses the *named* level of the *argument* rather the level of the result (in this case α rather than 0). The names of the levels will be used to generate the coercions discussed next.

2. does not perform the substitution for all the free variables, but rather, for a specific set of variables that come from the set X. As we will see in this section, the variables that are unaltered are essentially the ones that come from "outside the current environment classifier α". Given the typing of run, this also means that they are variables outside the scope of the current invocation of the run construct. This is essential for correctly dealing with the execution of open code, and will allow us to ensure that a term such as:

   ```
   .<fun x -> .~((run (a) .<%.<x>.>.)[b])>.
   ```

   reduces to

   ```
   .<fun x -> x>.
   ```

3. does not substitute with % but rather $\%_A$, which involves an additional set of escapes and brackets around its argument to make sure that the bracket that is being removed corresponds to the "right" environment. As we will explain at the end of this section, %s and matching brackets and escapes are computationally irrelevant, so this coercion is used only to capture the information needed to make these operations type theoretically correct.

$$\frac{}{i \overset{n}{\hookrightarrow} i} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \lambda x.e \\ e[x := e_2] \overset{0}{\hookrightarrow} e_3 \end{array}}{e_1\ e_2 \overset{0}{\hookrightarrow} e_3}$$

$$\frac{e_i \overset{n+}{\hookrightarrow} e_{i+2}}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_3\ e_4} \qquad \frac{e_1 \overset{n}{\hookrightarrow} e_2}{(\alpha)e_1 \overset{n}{\hookrightarrow} (\alpha)e_2} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} e_2}{e_1[\alpha] \overset{n+1}{\hookrightarrow} e_2[\alpha]} \qquad \frac{e_1 \overset{0}{\hookrightarrow} (\alpha)e_2}{e_1[\beta] \overset{0}{\hookrightarrow} e_1[\alpha := \beta]} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1\rangle^\alpha \overset{n}{\hookrightarrow} \langle e_2\rangle^\alpha}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \tilde{}e_2} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2\rangle^\alpha}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} (\alpha)\langle e_2\rangle^\alpha \\ e_2 \downarrow_\alpha^{[]} \overset{0}{\hookrightarrow} e_3 \end{array}}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} (\alpha)e_3} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+}{\hookrightarrow} \mathsf{run}\ e_2} \qquad \frac{e_1 \overset{n}{\hookrightarrow} e_2}{\%e_1 \overset{n+1}{\hookrightarrow} \%e_2}$$

Error-generating extension (used for proving type safety):

$$\frac{}{x \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_3 \quad e_3 \neq \lambda x.e}{e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_3 \neq \langle e_3\rangle^\alpha}{\tilde{}e_1 \overset{1}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_3 \quad e_3 \neq \langle e_4\rangle^\alpha}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv (\alpha)e_3}{e_1[\beta] \overset{0}{\hookrightarrow} \mathsf{err}}$$

Error-propagating extension:

$$\frac{e_2 \overset{0}{\hookrightarrow} \mathsf{err}}{e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \langle e_2\rangle^\alpha \\ e_2 \overset{0}{\hookrightarrow} \mathsf{err} \end{array}}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_i \overset{n+}{\hookrightarrow} \mathsf{err}}{e_1\ e_2 \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{\begin{array}{c} e_1 \overset{n+}{\hookrightarrow} \mathsf{err} \\ e \equiv \mathsf{run}\ e_1 \text{ or } \lambda x.e_1 \end{array}}{e \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\langle e_1\rangle^\alpha \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \mathsf{err}}$$

Figure 3: Big-step Operational Semantics for λ^α

Essentially all previous formulations of the demotion operation, whether explicit or implicit, traverse the whole term that is being demoted, only *counting* brackets, and not acting in a special way when a CSP point is encountered. This is a weakness, as brackets contained in a cross-stage persistent constant really bear no connection to the brackets that are currently being eliminated. The definition of demotion proposed here reflects this action in the case of %.

There are four basic *notions of reduction* for λ^α. The β reduction is almost standard. The restriction to stratified expressions and values of level 0 means that we cannot apply this reduction when any of the sub-terms involved contains an escape that is not matched by a surrounding bracket [62]. The W reduction is similar to β, but is for the classifier constructs. The escape reduction says when an escape can "cancel" a bracket. Similarly, the run reduction says when a run can cancel a bracket. Like previous definitions, the value restriction is essential for demotion to work: we can only eliminate a pair of brackets when all their corresponding escapes have been eliminated. Demotion is then used to make sure that the level 1 value is converted to an appropriate level 0 expression. This only involves reorganizing %s, which we will argue are computationally irrelevant at the end of this section.

The definitions of $\%\Gamma$, $|\Gamma|$, and Γ^A will be used in characterizing the typing behavior of demotion.

3.3.1 Subject Reduction

A desirable property of a typing is to withstand reduction:

THEOREM 2 (SUBJECT REDUCTION). *Let ρ range over $\{\beta, E, R, W\}$. Whenever $\Gamma \overset{A}{\vdash} e_1 : t$ and $e_1 \longrightarrow_\rho e_2$, then $\Gamma \overset{A}{\vdash} e_2 : t$.*

Establishing this theorem requires proving the key property for each of the reductions. For the β reduction, the main lemma needed is one that shows that typing is well-behaved under substitution:

LEMMA 1 (SUBSTITUTION). *1. $\Gamma \overset{A_1}{\vdash} e_1 : t_1$ and $\Gamma, x : t_1^{A_1}, \Gamma' \overset{A_2}{\vdash} e_2 : t_2$ implies $\Gamma, \Gamma' \overset{A_2}{\vdash} e_2[x := e_1] : t_2$.*

2. $\beta \in dom(\Gamma)$ and $\Gamma \overset{A}{\vdash} e : t$ implies $\Gamma[\alpha := \beta] \overset{A[\alpha:=\beta]}{\vdash} e[\alpha := \beta] : t[\alpha := \beta]$.

The reduction for escape is straightforward. The reduction for run requires characterizing the manner in which demotion affects typing. In the special case, demotion can be characterized as follows:

LEMMA 2 (SPECIAL DEMOTION). *$\Gamma, \alpha \overset{A,\alpha}{\vdash} v^1 : t$ implies $\Gamma, \alpha \overset{A}{\vdash} v^1 \downarrow_\alpha : t$*

This lemma gives us a way to take a value $v \in V^1$ typable at level A, α, and under an environment where the classifier α is at the top, and produce a term that has the same type, but at level A. This term will be computed by demoting v. Having such a well-defined set of conditions for "shifting a term down a level" is necessary for safe execution of code.

Both to carry out the proof for the above lemma, and to show that reduction "at any level" preserves typing, we need a more general (albeit less readable) result:

LEMMA 3 (DEMOTION). *$\Gamma, \alpha, \Gamma'^{A,\alpha} \overset{A,\alpha,A'}{\vdash} v^{|\alpha,A'|} : t$ implies $\Gamma, \alpha, \Gamma'^{A} \overset{A,A'}{\vdash} v^{|\alpha,A'|} \downarrow_{\alpha,A'}^{|\Gamma'^\alpha|} : t$*

To deal with the case for % at level α, we need a lemma:

Lemma 4. *$\Gamma, \alpha, \Gamma'^{A,\alpha}, \Gamma'' \overset{A'}{\vdash} e : t$ implies $\Gamma, \alpha, \Gamma'^{A}, \Gamma'' \overset{A'}{\vdash} e[\%\Gamma'^{\alpha}] : t$*

Where Γ^A and $\%\Gamma$ are defined in Figure 2. Note that the latter converts an environment into a substitution. The application of this substitution to a term e is denoted by $e[\%\Gamma]$.

3.4 Type Safety

Figure 3 presents the big-step semantics for λ^α. With the exception of classifier abstraction and instantiation and run, the rest of the rules are standard for multi-stage languages (see for examples [60, 42, 62, 7]). The rule for run is almost standard, except for the fact that it requires its argument to evaluate to a classifier abstraction rather than just a code fragment. Furthermore, it propagates this classifier abstraction in the return value.

Lemma 5. *1. $e \overset{n}{\hookrightarrow} e'$ then it is the case that $e' \in V^n$.*

2. $\Gamma^+ \overset{A}{\vdash} e : t$ and $e \overset{|A|}{\hookrightarrow} e'$ implies that $\Gamma^+ \overset{A}{\vdash} e' : t$.

If evaluation goes under some lambda abstraction, it must be at some named level α, A, and not simply a named level. To express this fact in typing judgments, we introduce the following auxiliary definition:

$$\Gamma^+ \quad \in \quad G^+ \quad ::= \quad [] \mid \Gamma^+, x : t^{\alpha,A} \mid \Gamma^+, \alpha$$

To establish type safety, we must extend our big-step semantics with rules for generating and propagating errors. Errors are represented by extending all expression families by a unique term err. The main problematic case for multi-stage languages is that evaluation at level 0 should not encounter a free variable. Thus, it is necessary to include this as an error-generating case in the augmented semantics.

Theorem 3. *$\Gamma^+ \overset{A}{\vdash} e : t$ and $e \overset{|A|}{\hookrightarrow} e'$ then $e \not\equiv \mathsf{err}$.*

The proof uses the lemmas introduced for subject reduction, in addition to properties of the big-step semantics.

3.5 Erasure Semantics

As the reference point we use the untyped big-step and reduction semantics for MetaML [62]. The additional constructs introduced in this paper are designed to be computationally irrelevant. The sense in which they are irrelevant is captured by an erasure function from λ^α to λ^U (of [62]), and is defined as follows:

$$\|i\| \equiv i,\ \|x\| \equiv x,\ \|\lambda x.e\| \equiv \lambda x.\|e\|,\ \|e_1\ e_2\| \equiv \|e_1\|\ \|e_2\|,$$
$$\|(\alpha)e\| \equiv \|e\|,\ \|e[\alpha]\| \equiv \|e\|,\ \|\langle e\rangle^\alpha\| \equiv \langle\|e\|\rangle,\ \|\tilde{}e\| \equiv \tilde{}\|e\|,$$
$$\|\mathsf{run}\ e\| \equiv \mathsf{run}\ \|e\|,\ \|\%e\| \equiv \tilde{}((\lambda x.\langle x\rangle)\ \|e\|)$$

A basic equational theory can be built from the reflexive, symmetric transitive closure of the notions of reductions presented above. We expect to be able to justify the soundness of such an equational theory by lifting the equational theory for the underlying untyped calculus λ^U using the erasure function above. We use a standard definition of observational equivalence but which is indexed by levels. We leave this development to future work.

4. POLYMORPHISM

Because of the practical utility of automatic type inference, an important extension of the λ^α calculus is the one to Hindley-Milner polymorphism [12, 31]. In this setting, polymorphism is implicit, but is limited to definitions introduced by a let-construct. Thus, the term let $id = \lambda x.x$ in e will bind the polymorphic identity function with type $\forall\tau.\tau \to \tau$ in the body e whereas $(\lambda id.e)(\lambda x.x)$ will bind a monomorphic identity of type (say) $\mathsf{int} \to \mathsf{int}$ in e.

As $(\alpha)t$ is already a form of quantification at the level of types (albeit over classifiers rather than general types) it can interfere with polymorphism. The central problem has to do with the α-substitution on types, which is needed in the rule for α-instantiation:

$$\frac{\Gamma \overset{A}{\vdash} e : (\alpha)t \quad \Gamma \vdash \beta}{\Gamma \overset{A,\alpha}{\vdash} e[\beta] : t[\alpha := \beta]}$$

Substitution is problematic when t is (or contains) a type variable: This type variable may at a later point become instantiated with a type containing α. As a concrete example consider the following legal typings of the term $\lambda x.x[\beta]$:

$$\beta \vdash \lambda x.x[\beta] : ((\alpha)\mathsf{int}) \to \mathsf{int}$$
$$\beta \vdash \lambda x.x[\beta] : ((\alpha)\langle\mathsf{int}\rangle^\alpha) \to \langle\mathsf{int}\rangle^\beta$$

What could be a type that generalizes both of these types? If this problem is not addressed, then the system lacks principal types. A reasonable candidate is $\forall\tau.((\alpha)\tau) \to \tau$. But this type cannot be instantiated to the type in the second derivation above, as it would require the value of τ to depend on the locally-bound name α.

One solution to dealing with this problem would be to use function types, thus allowing type variables to depend on the context in which they are used. They would allow us to unify the two above mentioned types into one type:

$$\forall\tau.((\alpha)\tau[\alpha]) \to \tau[\beta]$$

But with general type functions it is likely that type inference would become problematic. Thus we consider here a less standard approach that allows us to prove type safety for an expressive language, and which seems likely to enjoy reasonable inference properties. This approach explicitly delays the α-substitutions in type schemes until type variables are instantiated, at which point the α-substitutions can be resumed. The type scheme for the term in the derivations above would be $\forall\tau.((\alpha)\tau) \to (\tau(\alpha := \beta))$. Instantiating $\langle\mathsf{int}\rangle^\alpha$ for τ would give the type $((\alpha)\langle\mathsf{int}\rangle^\alpha) \to \langle\mathsf{int}^\beta\rangle$.

The type schemes need to remember any α-substitution so we extend the syntax for types with delayed α-substitutions. As α-substitutions can be pushed all the way down to type variables, we will only annotate type variables: $\tau(\theta)$ where $\theta = \{\alpha_i := \alpha'_i\}$ is a set of α-substitutions.

The syntax and type system for polymorphic λ^α are presented in Figure 4. The main change to the simply typed version of the type system is that environments carry type schemes instead of types. In the variable rule type schemes are instantiated according to the $\sigma \overset{\Gamma}{\succ} t$ relation. The abstraction rule binds the variable to a monomorphic type: $\forall().t_1$ which has only t_1 as an instance. On the other hand the rule for let generalizes the type t_1 over all free type variables in t_1 that are not in Γ.

$$\begin{array}{llcll}
\text{Type Schemes} & \sigma & \in & \Sigma & ::= \forall\tau_1,\dots,\tau_n.t \\
\text{Types with Variables} & t & \in & T^{\tau} & ::\overset{\pm}{=} \tau(\theta) \\
\alpha\text{-Substitution} & \theta & \in & R & ::= \alpha_1 := \beta_1,\dots,\alpha_n := \beta_n \\
\text{Expressions} & e & \in & E^{\tau} & ::\overset{\pm}{=} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}$$

Auxiliary definitions:

$$\frac{\Gamma \vdash \overline{t'}}{\forall\overline{\tau}.t \overset{\Gamma}{\succ} t\{\overline{\tau := t'}\}} \qquad \frac{\{\tau_1,\dots,\tau_n\} = \mathrm{FTV}(t)\setminus\mathrm{FTV}(\Gamma)}{\mathrm{close}(t,\Gamma)\equiv \forall\tau_1,\dots,\tau_n..t}$$

Type System:

$$\frac{\Gamma \overset{A}{\vdash}}{\Gamma \overset{A}{\vdash} i : \mathsf{int}} \qquad \frac{\Gamma(x)=\sigma^A \quad \sigma \overset{\Gamma}{\succ} t \quad \Gamma\overset{A}{\vdash}}{\Gamma \overset{A}{\vdash} x : t} \qquad \frac{\Gamma, x:\forall().t_1^A \overset{A}{\vdash} e : t_2}{\Gamma\overset{A}{\vdash}\lambda x.e : t_1\to t_2} \qquad \frac{\Gamma\overset{A}{\vdash} e_1 : t_1\to t_2 \quad \Gamma\overset{A}{\vdash} e_2 : t_1}{\Gamma\overset{A}{\vdash} e_1 e_2 : t_2}$$

$$\frac{\Gamma,\alpha\overset{A}{\vdash} e:t}{\Gamma\overset{A}{\vdash}(\alpha)e:(\alpha)t} \qquad \frac{\Gamma\overset{A}{\vdash} e:(\alpha)t \quad \Gamma\vdash\beta}{\Gamma\overset{A}{\vdash} e[\beta]: t\{\alpha:=\beta\}} \qquad \frac{\Gamma\overset{A,\alpha}{\vdash} e:t\quad\Gamma\vdash\alpha}{\Gamma\overset{A}{\vdash}\langle e\rangle^\alpha:\langle t\rangle^\alpha}$$

$$\frac{\Gamma\overset{A}{\vdash} e:\langle t\rangle^\alpha}{\Gamma\overset{A,\alpha}{\vdash}\ \tilde{}e:t} \qquad \frac{\Gamma\overset{A}{\vdash}e:t \quad \Gamma\overset{A,\alpha}{\vdash}}{\Gamma\overset{A,\alpha}{\vdash}\ \%e:t} \qquad \frac{\Gamma\overset{A}{\vdash} e:(\alpha)\langle t\rangle^\alpha}{\Gamma\overset{A}{\vdash}\mathsf{run}\ e:(\alpha)t} \qquad \frac{\Gamma\overset{A}{\vdash} e_1:t_1 \quad \Gamma, x:\mathrm{close}(t_1,\Gamma)^A\overset{A}{\vdash} e_2:t_2}{\Gamma\overset{A}{\vdash}\mathsf{let}\ x=e_1\ \mathsf{in}\ e_2:t_2}$$

Evaluation. The same rules as for $\cdot \overset{i}{\hookrightarrow} \cdot$ in Figure 3:

$$\frac{e_2[x:=e_1]\overset{0}{\hookrightarrow} e_4}{\mathsf{let}\ x=e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} e_4} \qquad \frac{e_1\overset{n+}{\hookrightarrow} e_3 \quad e_2\overset{n+}{\hookrightarrow} e_4}{\mathsf{let}\ x=e_1\ \mathsf{in}\ e_2 \overset{n+}{\hookrightarrow} \mathsf{let}\ x=e_3\ \mathsf{in}\ e_4}$$

Figure 4: Polymorphic λ^α

The most subtle change to the α-substitution is in the α-instantiation rule. It is replaced with a non-standard substitution operation $t\{\alpha := \beta\}$ that delays the substitution on the type-variables in t. This in turn implies that type instantiation should resume the delayed substitutions. This is done by defining the $\overset{\Gamma}{\succ}$ relation using a non-standard type substitution $t\{\tau := t'\}$. Examples of the two non-standard substitutions are:

- $(\langle\tau\rangle^\alpha \to \tau)\{\alpha := \beta\} = \langle\tau(\alpha := \beta)\rangle^\beta \to \tau(\alpha := \beta)$
- $(\langle\tau(\alpha := \beta)\rangle^\beta \to \tau(\alpha := \beta))\{\tau := \langle\mathsf{int}\rangle^\alpha\}$
  $= \langle\langle\mathsf{int}\rangle^\beta\rangle^\beta \to \langle\mathsf{int}\rangle^\beta$
- $(\langle\tau\rangle^\alpha \to \tau)\{\tau := \langle\mathsf{int}\rangle^\alpha\} = \langle\langle\mathsf{int}\rangle^\alpha\rangle^\alpha \to \langle\mathsf{int}\rangle^\alpha$
  $(\langle\langle\mathsf{int}\rangle^\alpha\rangle^\alpha \to \langle\mathsf{int}\rangle^\alpha)\{\alpha := \beta\} = \langle\langle\mathsf{int}\rangle^\beta\rangle^\beta \to \langle\mathsf{int}\rangle^\beta$

Notice the difference between the meta-level α-substitution $t\{\theta\}$ and the object-level α-substitution $\tau(\theta)$.

The primary property that demonstrates that this approach is sensible is this:

Lemma 6. $t\{\theta\}\{\overline{\tau := t'}\} = t\{\overline{\tau := t'}\}\{\theta\}$

This lemma states that the α-substitution and type substitution commute, and in effect delayed α-substitutions are correctly applied when type instantiation is performed. This lemma together with a host of less interesting lemmas about the different kinds of substitutions can be used to prove substitution, demotion, and subject reduction properties:

Lemma 7 (Substitution). $\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \overset{A}{\vdash} e_2 : t_2$ *and* $\Gamma_1 \vdash e_1 : t_1$ *and* $\{\overline{\tau}\}\cup\mathrm{FTV}(\Gamma_1) = \emptyset$ *implies* $\Gamma_1, \Gamma_2 \overset{A}{\vdash} e_2[x := e_1] : t_2$

For subject reduction, we extend our reductions with one rule:

$$\mathsf{let}\ x = v^0\ \mathsf{in}\ e^0 \quad \longrightarrow_L \quad e^0[x := v^0]$$

Theorem 4 (Subject Reduction). *Let* ρ *range over* $\{\beta, E, R, W, L\}$. *Whenever* $\Gamma \overset{A}{\vdash} e_1 : t$ *and* $e_1 \longrightarrow_\rho e_2$, *then* $\Gamma \overset{A}{\vdash} e_2 : t$.

5. EMBEDDINGS OF OTHER SYSTEMS

We present embeddings for the two main calculi representing the two main approaches to modeling staging into λ^α.

5.1 Linear Temporal Logic

The embedding of $\lambda^{\bigcirc}$ [14] into λ^α is direct. We assume that the base type b is int. We pick an arbitrary classifier α, and define the embedding as follows:

$$[\![\mathsf{int}]\!] \equiv \mathsf{int}, \quad [\![\bigcirc t]\!] \equiv \langle[\![t]\!]\rangle^\alpha, \quad [\![t_1 \to t_2]\!] \equiv [\![t_1]\!] \to [\![t_2]\!]$$
$$[\![n]\!] \equiv \alpha^n, \quad [\![x_i : t_i^{n_i}]\!] \equiv x_i : [\![t_i]\!]^{[\![n_i]\!]}$$
$$[\![x]\!] \equiv x, \quad [\![\lambda x.e]\!] \equiv \lambda x.[\![e]\!], \quad [\![e_1\ e_2]\!] \equiv [\![e_1]\!]\ [\![e_2]\!]$$
$$[\![\mathsf{next}\ e]\!] \equiv \langle[\![e]\!]\rangle^\alpha, \quad [\![\mathsf{prev}\ e]\!] \equiv \tilde{}[\![e]\!]$$

This embedding preserves typability in the following sense:

LEMMA 8. $\Gamma \overset{n}{\vdash}_{\bigcirc} e : t$ *implies* $(\alpha, [\![\Gamma]\!]) \overset{[\![n]\!]}{\vdash} [\![e]\!] : [\![t]\!]$

Note that only a single classifier is needed to embed a $\lambda^{\bigcirc}$ program.

The translation $[\![e]\!]$ maps syntactic constructs to syntactic constructs that have the same big-step semantics, so it is obvious that there is a lock-step simulation. However, it should be noted that notions of equivalence can differ between $\lambda^{\bigcirc}$ and λ^{α}: In the absence of a run construct, there is no context that can distinguish between $\mathsf{next}\ \lambda x.x$ and $\mathsf{next}\ \lambda x.\bot$, but in the target language these terms can be run (and then applied), and we can observe a difference.

5.2 Modal Logic

The primary strength of calculi working only with closed code is that there is no danger of running open code simply because there is none. Closed code calculi have traditionally been based on the modal logic S4 with the logic operator $\Box t$ denoting code of type t.

The λ^{α} calculus cannot express that a code fragment is closed—only that it is closed with respect to an environment α via the type $(\alpha)\langle t\rangle^{\alpha}$. It is a natural question whether this notion of closedness is enough to allow for an embedding of S4 into λ^{α}.

We consider an S4 calculus [16, Section 4.3] that is fairly close to the Kripke-model of S4 in that it has a stack of environments representing knowledge in different sets of worlds. Furthermore the use of the box and unbox constructs is similar to that of the $\langle - \rangle$ and $\tilde{}\,-$ construct, so an easy embedding seems feasible.

The embedding on types should map $\Box t$ to $(\alpha)\langle t'\rangle^{\alpha}$ to piggy-back on λ^{α}'s notion of α-closed code:

$$[\![\Box t]\!] \equiv (\alpha)\langle [\![t]\!]\rangle^{\alpha}, \quad [\![t_1 \to t_2]\!] \equiv [\![t_1]\!] \to [\![t_2]\!], \quad [\![\mathsf{int}]\!] \equiv \mathsf{int}$$

The translation of the term $\mathsf{box}\ e$ becomes $(\alpha)\langle e'\rangle^{\alpha}$, but when it comes to translating $\mathsf{unbox}_n\ e$, an α' is needed to instantiate $(\alpha)\langle t\rangle^{\alpha}$. The role of $\mathsf{unbox}_n\ e$ (for $n > 0$) is to splice in e into the surrounding code, and as all code is closed, it also means linking e's environment (albeit empty) to the environment of the surrounding code construction. Therefore the embedding on S4-terms must maintain a list of αs coming from surrounding α-quantifications:

$$\begin{array}{rcl} [\![\mathsf{box}\ e]\!]_A & \equiv & (\alpha)\langle [\![e]\!]_{A,\alpha}\rangle^{\alpha} \\ [\![\mathsf{unbox}_0\ e]\!]_{A,\alpha} & \equiv & (\mathsf{run}\ [\![e]\!]_{A,\alpha})[\alpha] \\ [\![\mathsf{unbox}_{n+1}\ e]\!]_{A,\alpha,\alpha_{n+1},\ldots,\alpha_1} & \equiv & \%^n(\tilde{}([\![e]\!]_{A,\alpha}[\alpha])) \end{array}$$
$$[\![i]\!]_A \equiv i, \quad [\![x]\!]_A \equiv x$$
$$[\![\lambda x.e]\!]_A \equiv \lambda x.[\![e]\!]_A, \quad [\![e_1 e_2]\!]_A \equiv [\![e_1]\!]_A[\![e_2]\!]_A$$

The translation of unbox_n depends on the subscript n. When $n > 0$ the term unbox_n corresponds to $\tilde{}\,-$, but if $n > 1$ it also digs into the environment stack to get code from previous stages, thus the need for the sequence of %s. On the other hand unbox_0 corresponds to running the code, and is translated into the λ^{α} version of run. Note however, that run leaves an unneeded α-quantifier, which has to be removed. The embedding of S4 environments (not stacks) is:

$$[\![x_1 : t_1, \ldots, x_n : t_n]\!] \equiv x_1 : [\![t_1]\!], \ldots, x_n : [\![t_n]\!]$$

Then the embedding of S4 into λ^{α} can be formulated as:

LEMMA 9. $\Gamma_0; \ldots; \Gamma_n \vdash e : t$ *implies* $\alpha_0, [\![\Gamma_0]\!], \ldots, \alpha_n, [\![\Gamma_n]\!] \overset{\alpha_1,\ldots,\alpha_n}{\vdash} [\![e]\!]_{\alpha_0,\ldots,\alpha_n} : [\![t]\!]$

This formulation of S4 was initially not presented with a direct operational semantics, but rather, via an interpretation into another "explicit calculus" [15]. A reduction semantics is presented in more recent work [16]. We expect that the reduction semantics can be used to derive a deterministic operational semantics, which can then be used for formally establishing a simulation result.

6. RELATED WORK

Multi-stage programming is a paradigm that evolved out of experience with partial evaluation [20, 35, 30, 34, 10] and runtime code generation systems [2, 28, 17, 18, 29]. Research in this area aims at providing the programmer with constructs that can be used to attain the performance benefits of the above mentioned techniques. These languages are also a natural outgrowth of two-level [48, 27] and multi-level languages [23, 24, 25, 15, 14], which where initially conceived to study the semantic foundations and efficient implementation techniques for both partial evaluation and runtime code generation. Recently, multi-stage languages have also been used to develop type systems for expressive, generative macros [21].

The use of an abstract quantifier was inspired by the recent work on FreshML [55, 43], parametricity [56, 65, 1], the typing of `runST` [37, 58, 41], and work on integrating system F with Hindley-Milner type systems [22]. A formal account of these connections remains an interesting open question.

Nanevski's recent work on the combination of $\lambda^{\Box}$ with type-safe intensional analysis is a promising way to give $\lambda^{\Box}$ a semantics closer to the calculi discussed in this paper [43]. It seems possible, however, that this may be primarily due to the power of intensional analysis (which should, in principle, allow the programmer to express arbitrary transformations). As with the explicit environment typing approach discussed in the introduction, types in Nanevski's system could in principle grow linearly in the number of free variables. There is currently no notion of polymorphism to deal with this issue in the context of Nanevski's system. Classifiers could be a useful approach to providing a such a mechanism.

Early attempts at developing a reduction semantics for untyped multi-stage languages involved a construct that can be interpreted as an explicit CSP constant (the "bubble" of λ^{T} of [60]), and included a number of reductions that propagate this construct "outward" in terms. The non-local nature of the demotion function presented here suggests that, while such notions of reduction could still be useful in implementations based on untyped calculi, they maybe incompatible with the type theory of multi-stage languages.

Michael Florentin Nielsen has been investigating extensions of Davies and Pfenning's modal calculus that provide a more direct remedy to the symbolic evaluation question. The full presentation is beyond the scope of this paper, but current results indicate that λ^{α} provides a lighter notation than this approach because it avoids the need to explicitly list the name and type of each variable potentially free in each code fragment. Another difficulty that λ^{α} seems to avoid is that the open code type allows inserting code fragments made in one environment into other environments. This implicit form of polymorphism must be made explicit in systems that Nielsen has developed based on Davies and Pfenning's modal calculus, unless an additional notion of polymorphism (possibly similar to ρ-polymorphism) is introduced. Note that even in λ^α there are no polymorphic variables that deal with environments: only environment classifiers, which in a sense range over whole *classes* of environments, and which are never instantiated to concrete instances. We continue to pursue a modal calculus, but we expect that its primary applications will be in providing better understanding of the semantics and possibly implementation techniques, and not necessarily in actual programming.

Davies[3] considered a similar approach to the one presented here. His work is directly based on natural deduction rules for two logical operators: "universal world quantification" and "truth in a particular world". The resulting language has some differences from the one presented here, but the basic idea seems related. One notable difference from our language is that there was no separate run construct (because the typing rule for run involves two type constructors, while each natural deduction rule should only refer to a single logical operator). Instead, there was a single (non-variable) world called "root", and the evaluation of "code" occurred whenever this world was substituted for a world variable.

7. CONCLUSIONS AND FUTURE WORK

We have presented a new approach to typing multi-stage languages, and showed how it can be used to embed previous approaches. We also showed that the approach is still sound in the presence of Hindley-Milner polymorphism. Key insights that come out of this work relate to subtleties involved in specifying a demotion lemma for an expressive type system. These details, in turn, provide concrete insights into the interplay between CSP and the presence of the run construct in a multi-stage language.

Currently, we have built only prototype implementations for the system presented here. We expect that the notion of classifiers will provide a natural mechanism to allow us to safely store and communicate open values, which to date has not been possible [8, 6, 7]. A crucial next step will be establishing the feasibility of type inference in the presence of environment classifiers. It will be interesting to see if the delayed substitutions approach developed in this paper is more helpful than using a more general approach, such as type functions. An overall goal will be to minimize type annotations, but not necessarily to eliminate the need for explicit world introduction and elimination in programs. Once these questions are resolved, a system based on λ^α will be introduced into the MetaOCaml public release. Implementing the system in the context of a full-fledged language will be essential for validating its practical expressivity.

The development presented here has two features potentially relevant to dependent types. First, because of the presence of classifiers, we have treated environments as ordered, and this was not problematic. Because ordered environments are a common feature of dependently typed languages, this may suggest that integration with dependently typed languages maybe be seamless. Second, also because of classifiers, we perform substitution on environments. This gives rise to a need for careful treatment of types in the Hindley-Milner polymorphic setting, but type safety still goes through. It maybe that some features of the present system maybe useful in the question of integrating dependent types with Hindley-Milner polymorphism.

Acknowledgments: Discussions with John Launchbury, Erik Meijer, Eugenio Moggi and Tim Sheard contributed greatly to the genesis of the ideas developed here. Cristiano Calcagno, Rowan Davies, and Eugenio Moggi provided valuable feedback on a draft of this paper. The reviewers provided many comments that improved the presentation greatly. Luke Hoban, Richard Nathan Linger and Emir Pašalić eliminated many typos in the final manuscript.

[3]Rowan Davies. Personal communication, Sept. 4th, 2002.

8. REFERENCES

[1] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 157–170. ACM New York, NY, 1993.

[2] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, 1996.

[3] Don Batory. Refinements and product line architectures. In *[61]*, pages 3–4, 2000.

[4] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[5] Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, 1999.

[6] Cristiano Calcagno and Eugenio Moggi. Multi-stage imperative languages: A conservative extension result. In *[61]*, pages 92–107, 2000.

[7] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2003. To appear.

[8] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, 2000. Springer-Verlag.

[9] Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, 15–17 January 1997.

[10] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[11] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in OPerating systems. In *Partial Evaluation and Semantics-Based*

Program Manipulation, Copenhagen, Denmark, June 1993, pages 44–46. ACM, 1993.

[12] Luís Damas and Robin Milner. Principal type schemes for functional languages. In *9th ACM Symposium on Principles of Programming Languages*. ACM, August 1982.

[13] Olivier Danvy, Karoline Malmkjaer, and Jens Palsberg. Eta-expansion does the trick. Technical Report RS-95-41, University of Aarhus, Aarhus, 1995.

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

[15] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.

[16] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[17] Dawson R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the Conference on Programming Language Design and Implemantation*, pages 160–170, New York, 1996. ACM.

[18] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 131–144, St. Petersburg Beach, 1996.

[19] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. In *[61]*, pages 108–128, 2000.

[20] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers and Control*, 2(5):45–50, 1971. [Via [33]].

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

[22] Garrigue and Remy. Semi-explicit first-class polymorphism for ML. *INFCTRL: Information and Computation (formerly Information and Control)*, 155, 1999.

[23] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[24] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

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

[26] Carsten K. Gomard. Partial type inference for untyped functional programs. In *ACM Conference on Lisp and Functional Programming*, 1990.

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

[28] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, 1997.

[29] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[30] Torben Amtoft Hansen, Thomas Nikolajsen, Jesper Larsson Träff, and Neil D. Jones. Experiments with implementations of two theoretical constructions. In *Lecture Notes in Computer Science 363*, pages 119–133. Springer Verlag, 1989.

[31] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.

[32] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[33] Neil D. Jones. Mix ten years later. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 24–38. ACM, 1995.

[34] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[35] Neil D. Jones, Peter Sestoft, and Harald Sondergraard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[36] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components II: Binary-level components. In *[61]*, pages 28–50, 2000.

[37] John Launchbury and Simon L. Peyton Jones. State in haskell. *LISP and Symbolic Computation*, 8(4):293–342, 1995. pldi94.

[38] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 108–118, N.Y., January 19–21 2000. ACM.

[39] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://cs-www.cs.yale.edu/homes/taha/MetaOCaml/, 2001.

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

[41] Eugenio Moggi and Amr Sabry. Monadic encapsulation of effects. *Journal of Functional Programming*, 2001.

[42] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[43] Aleksander Nanevski. Meta-programming with names and necessity. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[44] Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, 1985.

[45] Flemming Nielson. Correctness of code generation from a two-level meta-language. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP 86)*, volume 213 of *Lecture Notes in Computer Science*, pages 30–40, Saarbrücken, 1986. Springer.

[46] Flemming Nielson. A formal type system for comparing partial evaluators. In D Bjørner, Ershov, and Jones, editors, *Proceedings of the workshop on Partial Evaluation and Mixed Computation (1987)*, pages 349–384. North-Holland, 1988.

[47] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.

[48] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[49] Flemming Nielson and Hanne Riis Nielson. Multi-level lambda-calculi: An algebraic description. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354. Berlin: Springer-Verlag, 1996.

[50] Flemming Nielson and Hanne Riis Nielson. A prescriptive framework for designing multi-level lambda-calculi. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 193–202, Amsterdam, 1997. ACM.

[51] Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.

[52] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–142. IEEE Computer Society Press, 1998.

[53] 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.

[54] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[55] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Programme Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[56] John C. Reynolds. Towards a theory of types structure. In *Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, New York, 1974.

[57] Claudio Russo. *Types for Modules*. PhD thesis, Edinburgh University, 1998.

[58] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing through staged type inference. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 289–302, 1998.

[59] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, 1997.

[60] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [53].

[61] Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

[62] Walid Taha. 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.

[63] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, 1998.

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

[65] Philip Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept. 1989*, pages 347–359. ACM, New York, 1989.

[66] Mitchell Wand. Complete type inference for simple objects. In *Second Annual IEEE Symp. on Logic in Computer Science*, 1987.

[67] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.