Environment Classifiers

Extended Abstract

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

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

ABSTRACT

This paper proposes a new approach to typing multi-stage languages based a notion of *environment classifiers*. This approach involves explicit but lightweight tracking of the origination environment for future-stage computations in the types. Classifiers induce a notion of *classification* that is less restrictive than the previously proposed notions of closedness. This allows for a more expressive typing of the "run" construct, and for a unifying account of typed multi-stage programming. It also provides concrete new insights into the notion of levels and in turn into the notion of cross-stage persistence. The notion of classifiers extends to the Hindley-Milner polymorphic setting.

Keywords

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

1. INTRODUCTION

At the untyped level, multi-stage languages are a unifying framework for the essence of partial evaluation, program generation, runtime code generation, and generative macro-systems. 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 [40, 42, 43, 47, 44, 41, 45, 46] on one hand, and Gomard and Jones [31, 23, 24, 30] on the other, the question of whether these languages should support either closed code or open code has been a unresolved. 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.

*Work done while visiting Yale University

†Supported in part by NSF ITR-0113569. This work was done at Yale University.

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 [34] 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 simply invokes the power function whenever 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. Rather, brackets indicate that we are constructing a future stage computation, and an escape indicate 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 also performed immediately. The `run` on the last line invokes the compiler on the resulting code fragment, and incorporates into the runtime system.

1.2 When is Code Executable?

[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.

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. For example, in the expression

```
.<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 the early approaches multi-level languages seem to treat code either as always being open or always being closed [36]. 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 [12]. As illustrated above, the brackets and escapes can be used in to annotate λ-abstractions so as to force symbolic computations. This type system, however, 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 [13]. 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>.` [38]. 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 [38, 4, 7, 5, 6]. 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 closed code fragment and return code value. But not all open code, however, is unsafe to run. While this approach is acceptable for executing single-stage programs (which do not contain future stage variables), 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 will consider (c.f [54]): 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 could be the use of dependently typed inductive datatypes (as is done for object languages programs in the recent work on Meta-D [49]). 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 [50, 39], 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 [56, 61], which is a key benefits of multi-stage languages.

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 [8], implicit parameters [33], module systems [35, 52], explicit substitutions [3], and program analysis [17]. The general flavor of such an approach can be illustrated as follows:

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

Close inspection reveals that this approach may not be as appealing as it may seem 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 [11], 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 needed. This function would

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

Types and Environments:

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

Terms:

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

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

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

need to take a value of type $\langle t_1 \rangle^e \to \langle t_2 \rangle^e$ and returns 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 [60], 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 Contributions

The essential 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 2.

The proposed approach sheds **new insights into the notion of cross-stage persistence**. *Cross-stage persistence* [58] 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. In particular, there are two forms of cross-stage persistence: Implicit and explicit. With the implicit approach, CSP is only admitted on variables through a rule such as:

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

which compares the bracket-depth at which a variable is bound (m) and makes sure that it is less than that at which the variable is used (n). It must then be separately demonstrated the there is also an analogous notion of implicit CSP (called "promotion") that holds on terms [57, 38]. The implicit approach, therefore, allows less structure than what we show here to be possible and essential for executing open term. In particular, the implicit approach (and developments based on it [6]) considers a term to be cross-stage persistent if and only its variables can be made cross-stage persistence. With the explicit approach, there is a dedicated constructs 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 both presentations, consider the term:

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

which evaluates under the standard untyped big-step semantics for multi-stage languages (see for example [56]) to `.<fun x -> x>.`. Note that `x` is surrounded by only one bracket when it is bound, and two brackets (that are not canceled by an escape) when it is used. If we have implicit cross-stage persistence, then 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 match the escape, and so, `x` is in fact surrounded immediately by the "right" brackets. An acceptable interpretation, however, is that the argument to run is a term `.<...>.` containing a cross-stage persistent constant `.<x>.`. We can express this observation equationally as follows:

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

in general for the notion of typed equality $\approx$ in the type system that we are interested. The notion of classifiers will allow a finer degree of control on the use of cross-stage per-

sistence than was previously possible. The technical reasons for why classifiers are useful for dealing with this issue in the context of the details of the demotion lemma, and careful comparison with similar developments in the literature. The demotion lemma is the cornerstone in the argument for why running a given code fragment is safe.

The soundness of Hindley-Milner polymorphism in the presence of statically typed brackets and escaped does not seem to have been addressed in the literature (c.f. [6]). Dynamically typed versions of brackets, escape, and run have been used to introduce a notion of staged type inference [53]. 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 3 demonstrates the **soundness of typed multi-stage programming in the presence of Hindley-Milner polymorphism**.

Section 4 shows how a type system based on this approach provides a **unifying account of typed multi-stage programming**. The resulting type system has no more constructs than the previous systems.

Selected proofs are included in the appendix which will be published on-line in an extended version of this paper.

2. ENVIRONMENT CLASSIFIERS

In this section we introduce the concept of environment classifiers by means of a multi-stage calculus called λ^{α}. After presenting its type system and semantics, we demonstrate its key properties, such as type safety. We also compare various features of λ^{α} to previous proposals.

2.1 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 an 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 notion of traditional notion of levels in multi-stage languages [12, 58, 56], 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_2 with $t_1 \rightarrow t_2$. Next is α-closed terms $(\alpha)t$, read "α-closed t". 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 $\bigcirc t$ [12] (that latter also being essentially the open code type used in previous proposals for type systems for MetaML [57, 38, 54].) 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 carrying 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 a environment declaration α. Addition 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 $(\alpha)t$ (read "α-closed t") and instantiation $t[\alpha]$ (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 environment. The next three constructs are essentially the standard constructs of MetaML: Brackets $\langle e \rangle^{\alpha}$, escape $\tilde{}e$, and run $\mathsf{run}\ e$. Having an α annotations, however, is not standard. While this annotation has no effect on the operations semantics, it is a declaration of the environment this code fragments should be associated with. In practice, the last α in the environment seems to be the most appropriate. Thus, we take the last alpha in the environment as the default, and require the user to provide explicitly only when it is different from the default. Next is an explicit construct for cross-stage persistence $\%e$. In MetaML, cross-stage persistence is limited to variables (in the typing rules). In the current proposal, however, cross-stage persistence will be allowed on terms.

As mentioned before, environments are ordered. There are additional requirements on the structure of environments. A type is well-formed under 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 introduce in the environment to its left.

2.2 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 also recorded in the environment.

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 rule requires that α was not introduced previously by the environment Γ. The well-formedness condition 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 environments that α classifies. The rule for α instantiation allows us to replace 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 system 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

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

Demotion (and Auxiliary Definitions):

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

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

$$
\%_\alpha e \equiv \%e, \quad \%_{A,\alpha}e \equiv \tilde{}(\%_A\langle e\rangle^\alpha), \quad \%(\Gamma,\alpha) \equiv \%\Gamma, \quad \%(\Gamma, x:t^A) \equiv (\%\Gamma, x := \%_A x)
$$

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

Notions of Reduction:

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

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

at level A, α only allows the incorporation of code fragments of type $\langle t\rangle^\alpha$.

The rule for cross-stage persistence 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 cross-stage persistence. Finally, the rule for run allows us to execute a code fragment of type $\langle t\rangle^\alpha$ only if it is α-closed.

2.3 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 the key operation need 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 cross-stage persistence, this operation is syntactic identity [38]. The definition here is similar to previous definitions used in calculi with explicit cross-stage persistence , with key difference: 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 differs in a number of ways:

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

2. It 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 values that are unaltered are essentially the values that come from "outside the current environment classifier α". 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>.>.>.
   ```

   reduces to

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

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

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

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

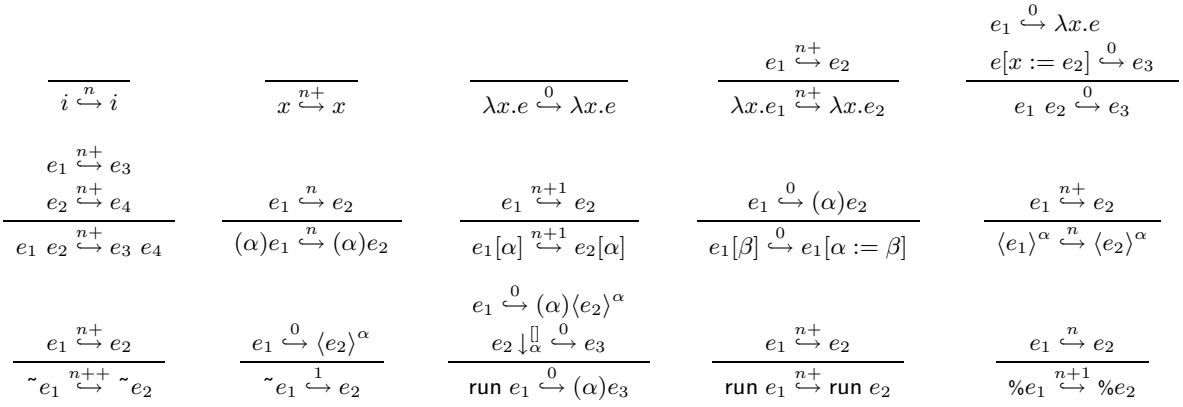

Figure 3: Big-step Operational Semantics for λ^α

an escape that is not matched by a surrounding bracket. 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 an appropriate level 0 expressions. That this only involves reorganizing %s, which will be shown to be 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.

2.3.1 Subject Reduction

A desirable property of a typing is to withstand reduction:

THEOREM 1 (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 $\Gamma, x : t^{A_1}, \Gamma' \overset{A_1}{\vdash} e_2 : t_2$ implies $\Gamma, \Gamma' \overset{A_1}{\vdash} e_2[x := e_1] : t_2$.*

2. $\Gamma, \alpha \overset{A}{\vdash} e : t$ implies $\Gamma \overset{A}{\vdash} e[\alpha := \beta] : t[\alpha := \beta]$.

The reduction for escape is straightforward. The reduction for run requires characterizing the manner in which demotion affects typing:

LEMMA 2 (DEMOTION). $\Gamma, \alpha, \Gamma'^{A,\alpha} \overset{A,\alpha,A'}{\vdash} v^{|\alpha,A'|} : t$ *implies* $\Gamma, \alpha, \Gamma'^{A} \overset{A,A'}{\vdash} v^{|\alpha,A'|}\downarrow^{|\Gamma'^{\alpha}|}_{\alpha,A'} : t$

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

LEMMA 3. $\Gamma, \alpha, \Gamma'^{A,\alpha}, \Gamma'' \overset{A'}{\vdash} e : t$ *implies* $\Gamma, \alpha, \Gamma'^{A}, \Gamma'' \overset{A'}{\vdash} e[\%\Gamma'^{\alpha}] : t$

2.4 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 [54, 38, 56, 6]). 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 4. *1. $e \overset{n}{\hookrightarrow} e'$ then it is the case that $e' \in V^n$.*

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

If evaluation goes under some lambda abstraction, it must be at some named level α, A, and not a imply 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 particular important to include this as an error-generating case in the augmented semantics. The full definition of this semantics is included as Figure 6 in the Appendix.

THEOREM 2. $\Gamma^+ \overset{A}{\vdash} e : t$ *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.

2.5 Equational Theory

A basic equational theory can be built from the reflexive, symmetric transitive closure of the notions of reductions presented above. We justify the soundness of such an equational theory by lifting the equational theory for an underlying untyped calculus. We use a standard definition of observational equivalence but which is indexed by levels. The complete definition of this notion can be found in AppendixB.1.

THEOREM 3 (SOUNDNESS OF CBV λ^α REDUCTIONS). *For any* $\Gamma \overset{A}{\vdash} e_1, e_2 : t$

$$e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx_{|A|} e_2.$$

This result follows from the fact that observations on level 0 expressions for λ^α are the same as those λ^0, and that the notions of reductions erase into precisely those of λ^U.

3. POLYMORPHISM

An interesting extension of the λ^α calculus is the one with Hindley-Miller polymorphism [10, 28], where polymorphism is limited to let-constructs: The term $\mathsf{let}\ id = \lambda x.x\ \mathsf{in}\ e$ 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 $(\alpha)t$ already is a kind of polymorphism, quantifying over environments instead of types, one might be concerned if the two kinds of polymorphisms might interfere, and in fact they do. The central problem has to do with the α-substitution on the type in the rule for α-instantiation:

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

This 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 $\lambda x.x[\beta]$:

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

Clearly the type scheme $\forall\tau.((\alpha)\tau) \to \tau$ is an incorrect generalization. Two solutions present themselves:

- The type system could restrict the legal instantiations of the type variable τ to only types with no free occurrences of α. The above type scheme could look like $\forall\tau_{\{\alpha\}}.((\alpha)\tau_{\{\alpha\}}) \to \tau_{\{\alpha\}}$, where $\tau_{\{\alpha\}}$ denotes any type where α is not free.

- The other solution delays the α-substitutions in type schemes to when type variables are instantiated at which point the α-substitutions can be resumed. The type scheme for the above example would look like $\forall\tau.((\alpha)\tau) \to (\tau(\alpha := \beta))$. Instantiating $\langle\mathsf{int}\rangle^\alpha$ for τ would give the type $((\alpha)\langle\mathsf{int}\rangle^\alpha) \to \langle\mathsf{int}^\beta\rangle$.

We will choose the latter as the former seems too restrictive: Many interesting application of the function $\lambda x.x[\beta]$ seem to be on values whose types would have α free in τ.

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 of the polymorphic λ^α is presented in Figure 4, the type system in Figure 5, and some axillary definitions for the type system in Figure 8 in the appendix. The main change to the simply typed version of the type system is that environments carries type schemes instead of types. In the variable rule type schemes are instantiated according to the $\sigma \succ t$ 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.

The most subtle change to the α-substitution 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 $\succ$ 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 5. $t\{\theta\}\{\overline{\tau := t'}\} = t\{\overline{\tau := t'}\}\{\theta\}$

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 (see appendix C for details):

LEMMA 6 (SUBSTITUTION).

$$\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \overset{A}{\vdash} e_2 : t_2 \quad \textit{and}$$
$$\Gamma_1 \vdash e_1 : t_1 \quad \textit{and} \quad \{\overline{\tau}\} \cup \mathrm{FTV}(\Gamma_1) = \emptyset$$
$$\Longrightarrow$$
$$\Gamma_1, \Gamma_2 \overset{A}{\vdash} e_2[x := e_1] : t_2$$

$$
\begin{array}{llcll}
\text{Type schemes} & \sigma & \in & \Sigma & ::= \forall\tau_1,\ldots,\tau_n.t \\
\text{Types with variables} & t & \in & T^\tau & ::= \ldots\text{(same as for } T\text{)} \\
 & & & & \mid \tau(\theta) \\
\alpha\text{-substitution} & \theta & \in & R & ::= \alpha_1 := \beta_1,\ldots,\alpha_n := \beta_n \\
\text{Expressions} & e & \in & E^\tau & ::= \ldots\text{(same as for } E\text{)} \\
 & & & & \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

Figure 4: Syntax for polymorphic λ^α. Refer to Figure 4 for T and E.

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

$$
\frac{\Gamma \overset{A}{\vdash} e_1 : t_1 \to t_2 \quad \Gamma \overset{A}{\vdash} e_2 : t_1}{\Gamma \overset{A}{\vdash} e_1 e_2 : t_2} \qquad \frac{\Gamma, \alpha \overset{A}{\vdash} e : t}{\Gamma \overset{A}{\vdash} (\alpha)e : (\alpha)t}\ \alpha \notin dom(\Gamma) \qquad \frac{\Gamma \overset{A}{\vdash} e : (\alpha)t \quad \Gamma \vdash \beta}{\Gamma \overset{A}{\vdash} e[\beta] : t\{\alpha := \beta\}} \qquad \frac{\Gamma \overset{A,\alpha}{\vdash} e : t \quad \Gamma \vdash \alpha}{\Gamma \overset{A}{\vdash} \langle e \rangle^\alpha : \langle t \rangle^\alpha}
$$

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

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

$$
\frac{e_2[x := e_1] \overset{0}{\hookrightarrow} e_4}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} e_4} \qquad \frac{e_1 \overset{i+1}{\hookrightarrow} e_3 \quad e_2 \overset{i+1}{\hookrightarrow} e_4}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{i+1}{\hookrightarrow} \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}
$$

Figure 5: Type system and evaluation for polymorphic λ^α.

LEMMA 7 (READABLE DEMOTION).

$$
\begin{array}{l}
\Gamma, \alpha \overset{A,\alpha}{\vdash} v^1 : t \\
\Longrightarrow \\
\Gamma, \alpha \overset{A}{\vdash} v^1 \downarrow_\alpha : t
\end{array}
$$

THEOREM 4 (SUBJECT REDUCTION).

$$
\begin{array}{l}
\Gamma \overset{A}{\vdash} e : t \quad \textit{and} \quad e \longrightarrow e' \\
\Longrightarrow \\
\Gamma \overset{A}{\vdash} e' : t
\end{array}
$$

The motivation for choosing the system with delayed α-substitution over the system with restricted polymorphism was to have more general types. In the restricted version the types $((\alpha)\mathsf{int}) \to \mathsf{int}$ and $((\alpha)\langle\mathsf{int}\rangle^\alpha) \to \langle\mathsf{int}\rangle^\beta$ can not be generalized to a type scheme, even though they are legal types of the same term. This begs the question whether the system in this section does have principal types. We leave this question to future work.

4. EMBEDDINGS OF OTHER SYSTEMS

In this section we present the embeddings for the two main calculi that represent the two main approaches to modeling staged computation into λ^α.

4.1 Linear Temporal Logic

The embedding $\lambda^\bigcirc$ [12] 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:

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

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 not 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.

4.2 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 has traditionally been based on the modal logic S4 with the logic operator $\Box t$ denoting the code type of 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.

One S4 calculus [14] is exhibited in Figure 7 in the appendix. Its formulation 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 ~− construct, so an easy embedding seems feasible.

The intended embedding on types should map $\Box t$ to $(\alpha)\langle t'\rangle^\alpha$ to piggy bag on λ^αs notion of α-closed code:

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

The translation of the term 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 es 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 containing α-quantifications:

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

The translation of unbox_n depend on the subscript n. When $n > 0$ the term unbox_n corresponds to ~−, 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:

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

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 not presented with a direct operational semantics, but rather, via an interpretation into another "explicit calculus" [13]. We leave formally establishing a simulation result for future work.

5. RELATED WORK

Multi-stage programming is a paradigm that evolved out of experience with partial evaluation [18, 31, 27, 30, 9] and runtime code generation systems [2, 25, 15, 16, 26]. 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 [44, 24] and multi-level languages [20, 21, 22, 13, 12], 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 types systems for expressive, generative macros [19].

The use of an abstract quantifier was inspired by the recent work on FreshML [50, 39], parametricity [51, 59, 1], and the typing of `runST` [32, 53, 37]. We omit a detail discussions of these connections for reasons of space.

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 [39]. 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 cross-stage persistence constant (λ^T of [54]), and included a number of reductions that propagate this construct "outwards" 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.

The first named author has been investigating extensions of Davies and Pfenning's modal calculus that provide a more direct remedy to the symbolic evaluation question. While the full presentation is beyond the scope of this paper, current results indicate that λ^α provides a lighter notation than this approach because it avoids the need to explicitly list the names and types 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 the systems that we have 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 we there are no polymorphic variables that deal with environments: only environment classifiers, which in some sense range over whole *classes* 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.

6. CONCLUSIONS AND FUTURE WORK

We have presented a new approach to typing multi-stage programming, and showed how it can be used to embed previous approaches to typed multi-stage programming. We also show that the approach also works in the presence of Hindley-Milner polymorphism. Key insights that come out of this work relate to the subtleties involved in specifying

a demotion lemma for an expressive type system. These details, in turn, provide concrete insights into the interplay between cross-stage persistence and the presence of the run construct in a multi-stage language.

Currently, we have only built prototype implementations for the system presented here. We expect that the notion of a classifiers will provide a natural mechanism to allow us to safely store and communicate open values, which to date has not been possible [7, 5, 6]. Once these results are established, a system based on λ^{α} will be introduced into the MetaOCaml public release.

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 polymorphism, but the development also goes through. It maybe that some features of the present system maybe useful in the question of integrating dependent types with Hindley-Milner polymorphism.

7. REFERENCES

[1] M. Abadi, L. Cardelli, and P.-L. 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] 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.

[4] 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.

[5] Cristiano Calcagno and Eugenio Moggi. Multi-stage imperative languages: A conservative extension result. In *[55]*, pages 92–107, 2000.

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. Submitted. Available as Technical Report CMU-CS-99-153, August 1999.

[15] 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 Press.

[16] 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.

[17] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. In *[55]*, pages 108–128, 2000.

[18] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers and Control*, 2(5):45–50, 1971. [Via [29]].

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

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

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

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

[31] 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.

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

[33] 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 Press.

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

[35] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[36] 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.

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

[38] 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.

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

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

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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 Press.

[47] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.

[48] 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.

[49] 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.

[50] A. M. Pitts and M. 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.

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

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

[53] 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.

[54] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [48].

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

[56] 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 Press.

[57] 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.

[58] 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 Press.

[59] 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 Press, New York, 1989.

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

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

APPENDIX

A. TYPE SAFETY

Figure 6 presents the extension of the big-step semantics for core λ^α with error generation and propagation cases. The proof proceeds by a straight forward induction on the height of the evaluation derivation. Substitution lemmas are used in the application cases, and the demotion lemma is used in the case of run.

B. EQUATIONAL THEORY

We will give the erasure semantics for λ^α in terms of a call-by-value (CBV) variant of λ^U [56]. Stratified expressions for λ^U are exactly the same as those for λ^α, except that λ^U does not have classifier abstraction, classifier instantiation, or an explicit construct for cross-stage persistence. We will denote the expression and value sets for λ^U by E^U and V_U, respectively. The big-step semantics for this language is exactly the same as the big-step semantics of λ^α for constructs above. The reduction semantics for λ^U is defined as follows:

DEFINITION 1 (CBV λ^U REDUCTIONS).

$$\begin{aligned}(\lambda x.e^0)\, v^0 &\longrightarrow_{\beta_U} e^0[x := v^0]\\ \tilde{}\langle e^0\rangle &\longrightarrow_{E_U} e^0\\ \mathsf{run}\,\langle e^0\rangle &\longrightarrow_{R_U} e^0.\end{aligned}$$

DEFINITION 2 (REDUCTION RELATION). *The CBN λ^U reduction relation $\longrightarrow \subseteq E \times E$ as the compatible extension of rewriting a term using any of the three λ^U notions of reduction. We write $\longrightarrow^*$ for its reflexive, transitive closure.*

Equivalence is defined as follows:

DEFINITION 3 (LEVEL 0 TERMINATION). $\forall e \in E^0$.

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

Error-generating extension:

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

Error-propagating extension:

$$\frac{e_2 \overset{0}{\hookrightarrow} \mathsf{err}}{e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \alpha \quad e_2 \overset{0}{\hookrightarrow} \mathsf{err}}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_{1/2} \overset{n+}{\hookrightarrow} \mathsf{err}}{e_1\ e_2 \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\langle e_1 \rangle \alpha \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\mathsf{run}\ e_1 \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} \mathsf{err}}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \mathsf{err}}$$

Figure 6: Big-step Operational Semantics extension for λ^{α}

$$\frac{}{\Psi; \Gamma \vdash x : t}\ t = \Gamma(x) \qquad \frac{\Psi; \Gamma, x : t_1 \vdash e : t_2}{\Psi; \Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \frac{\Psi; \Gamma \vdash e_1 : t_1 \to t_2 \quad \Psi; \Gamma \vdash e_2 : t_1}{\Psi; \Gamma \vdash e_1 e_2 : t_2}$$

$$\frac{\Psi; \Gamma; () \vdash e : t}{\Psi; \Gamma \vdash \mathsf{box}\ e : \Box t} \qquad \frac{\Psi; \Gamma \vdash e : \Box t}{\Psi; \Gamma; \Gamma_1; \ldots; \Gamma_n \vdash \mathsf{unbox}_n\ e : t}$$

Figure 7: Type system for S4

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

$$C \in \mathbb{C} := \quad [] \mid \lambda x.C \mid C\,e \mid e\,C \mid \langle C \rangle \mid \tilde{}C \mid \mathsf{run}\ C.$$

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

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

Theorem 5 (Soundness of CBV λ^U Reductions). $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.

$$e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx_n e_2.$$

This result is established using a standardization argument essentially identical to that used for CBN λ^U [56].

The definition of the erasure function from λ^{α} to λ^U is as follows:

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

First we note the sense in which the translation preserves values:

Lemma 10. $\Gamma \overset{A}{\vdash} v^{|A|} : t$ *implies* $\exists e' \in V^{|A|}.e' \approx_{|A|} \|v^{|A|}\|$.

The we can establish the following implication:

Lemma 11. *For any* $\Gamma \overset{A}{\vdash} e : t$*:*

1. $e \overset{n}{\hookrightarrow} e'$ *implies* $\exists e'' \approx_n \|e'\|.\|e\| \overset{n}{\hookrightarrow} e''$.
2. $\|e\| \overset{n}{\hookrightarrow} e''$ *implies* $\exists e'.\|e'\| \approx_n e''$ *and* $e \overset{n}{\hookrightarrow} e'$.

Proof. Forward implication: Most cases are straightforward. The key insight in the case of run is that demotion translates into no-ops because of the interpretation of ups. Backward implication is similar.

B.1 Equational theory for λ^{α}

Definition 6 (Level 0 Termination). $\forall e \in E^0$.

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

Definition 7 (Context). *A* Context *is an expression with exactly one hole* $[]$.

$$C \in \mathbb{C} := \quad [] \mid \lambda x.C \mid C\,e \mid e\,C \mid (\alpha)C \mid C[\alpha] \mid \\ \langle C \rangle \mid \tilde{}C \mid \mathsf{run}\ C \mid \%C.$$

Definition 8 (Observational Equivalence). *We define* $\approx_n \in E^n \times E^n$ *as follows:* $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.

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

C. POLYMORPHISM

This section contains the various lemmas used to prove subject reduction of the polymorphic λ^{α} calculus.

Lemma 12.

$$t\{\theta\}\{\theta'\} = t\{\theta' \circ \theta\}$$

Polymorphic type closing, $\mathrm{close}(t, \Gamma)$):

$$\mathrm{close}(t, \Gamma) = \forall \tau_1, \ldots, \tau_n . . t \quad \text{where} \quad \{\tau_1, \ldots, \tau_n\} = \mathrm{FTV}(t) \setminus \mathrm{FTV}(\Gamma)$$

Type scheme instantiation, $\sigma \succ t$:

$$\forall \overline{\tau}.t \succ t\{\overline{\tau := t'}\}$$

extended to type schemes ($\sigma \succ \sigma'$):

$$\sigma \succ \sigma' \quad \text{iff} \quad \sigma' \succ t \text{ implies } \sigma \succ t$$

Non-standard type substitution,$t\{\overline{\tau := t'}\}$:

$$\begin{array}{rcl}
\mathsf{int}\{\overline{\tau := t}\} & = & \mathsf{int} \\
(\tau_i(\theta))\{\tau_1 := t_1, \ldots, \tau_n := t_n\} & = & t_i\{\theta\} \\
(\tau(\theta))\{\tau_1 := t_1, \ldots, \tau_n := t_n\} & = & \tau(\theta) \quad \text{if } \tau \neq \tau_i \text{ for all } 1 \leq i \leq n \\
(t_1 \to t_2)\{\overline{\tau := t_3}\} & = & t_1\{\overline{\tau := t_3}\} \to t_2\{\overline{\tau := t_3}\} \\
\langle t_1 \rangle^{\alpha}\{\overline{\tau := t_2}\} & = & \langle t_1\{\overline{\tau := t_2}\} \rangle^{\alpha} \\
((\alpha)t_1)\{\overline{\tau := t_2}\} & = & (\alpha)(t_1\{\overline{\tau := t_2}\})
\end{array}$$

extended to type schemes ($\sigma\{\overline{\tau := t'}\}$):

$$(\forall \overline{\tau}.t)\{\overline{\tau' := t'}\} = \forall \overline{\tau''}.(t[\overline{\tau := \tau''}]\{\overline{\tau' := t'}\}) \quad \overline{\tau''} \text{ fresh}$$

Extended α-substitution, $t'\{\theta\}$:

$$\begin{array}{rcl}
\mathsf{int}\{\theta\} & = & \mathsf{int} \\
\tau(\theta')\{\theta\} & = & \tau(\theta \circ \theta') \\
(t_1 \to t_2)\{\theta\} & = & t_1\{\theta\} \to t_2\{\theta\} \\
\langle t \rangle^{\alpha}\{\theta\} & = & \langle t(\theta) \rangle^{\theta(\alpha)} \\
((\gamma)t)\{\theta\} & = & (\gamma)(t\{\theta\}) \quad \gamma \notin \theta
\end{array}$$

α-substitution lookup, $\theta(\alpha)$:

$$\begin{array}{rcl}
(\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n)(\alpha_i) & = & \beta_i \\
(\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n)(\alpha) & = & \alpha \quad \text{if} \quad \alpha \neq \alpha_i
\end{array}$$

Combination, $\theta \circ \theta'$, of α-substitution is any operation satisfying:

$$(\theta \circ \theta')(\alpha) = \theta(\theta'(\alpha))$$

Figure 8: Polymorphic type closing, type scheme instantiation, non-standard type substitution, extended α-substitution, and α-substitution combination.

PROOF. By a simple induction over t.

The following lemma also appears as Lemma 5 in section 3.

LEMMA 13.

$$t\{\theta\}\{\overline{\tau := t'}\} = t\{\overline{\tau := t'}\}\{\theta\}$$

PROOF. By induction over t.

- int

$$\mathsf{int}\{\theta\}\{\overline{\tau := t'}\} = \mathsf{int} = \mathsf{int}\{\overline{\tau := t}\}\{\theta\}$$

- $\tau_i(\theta')$

$$\tau_i(\theta')\{\theta\}\{\overline{\tau := t'}\} = \tau_i(\theta \circ \theta')\{\overline{\tau := t'}\} = t'_i\{\theta \circ \theta'\}$$
$$= t'_i\{\theta'\}\{\theta\} = \tau_i(\theta')\{\overline{\tau := t'}\}\{\theta\}$$

  using lemma 12.

- $t_1 \to t_2$

$$\begin{aligned}&(t_1 \to t_2)\{\theta\}\{\overline{\tau := t'}\}\\ &= t_1\{\theta\}\{\overline{\tau := t'}\} \to t_2\{\theta\}\{\overline{\tau := t'}\}\\ &\overset{IH}{=} t_1\{\overline{\tau := t}\}\{\theta\} \to t_2\{\overline{\tau := t}\}\{\theta\}\\ &= (t_1 \to t_2)\{\overline{\tau := t}\}\{\theta\}\end{aligned}$$

- $\langle t \rangle^\alpha$

$$\begin{aligned}&\langle t \rangle^\alpha\{\theta\}\{\overline{\tau := t'}\} = \langle t\{\theta\}\{\overline{\tau := t'}\}\rangle^{\theta(\alpha)}\\ &\overset{IH}{=} \langle t\{\overline{\tau := t}\}\{\theta\}\rangle^{\theta(\alpha)} = \langle t \rangle^\alpha\{\overline{\tau := t}\}\{\theta\}\end{aligned}$$

□

LEMMA 14. *If* $\{\overline{\tau'}\} \cap \{\overline{\tau''}\} = \emptyset$ *and* $\{\overline{\tau'}\} \cap \mathrm{FTV}(t'') = \emptyset$ *then*

$$t\{\overline{\tau' := t'}\}\{\overline{\tau'' := t''}\} = t\{\overline{\tau'' := t''}\}\{\overline{\tau' := t'\{\overline{\tau'' := t''}\}}\}$$

PROOF. By induction over t.

- $\tau'_i(\theta)$ when $\overline{\tau' := t'} = \tau'_1 := t'_1, \ldots, \tau'_n := t'_n$ and $1 \leq i \leq n$

$$\begin{aligned}&\tau'_i(\theta)\{\overline{\tau' := t'}\}\{\overline{\tau'' := t''}\}\\ &= \tau'_i\{\overline{\tau' := t'}\}\{\overline{\tau'' := t''}\}\{\theta\}\\ &= t'_i\{\overline{\tau'' := t''}\}\{\theta\}\\ &= \tau'_i\{\tau' := t'\{\overline{\tau'' := t''}\}\}\{\theta\}\\ &= \tau'_i\{\overline{\tau'' := t''}\}\{\overline{\tau' := t'\{\overline{\tau'' := t''}\}}\}\{\theta\}\\ &= \tau'_i(\theta)\{\overline{\tau'' := t''}\}\{\tau' := t'\{\overline{\tau'' := t''}\}\}\end{aligned}$$

  By using Lemma 5 and the fact that $\tau\{\theta\} = \tau(\theta)$.

- $\tau(\theta)$ when $\tau \notin \{\overline{\tau'}\}$

$$\begin{aligned}&\tau(\theta)\{\overline{\tau' := t'}\}\{\overline{\tau'' := t''}\}\\ &= \tau\{\overline{\tau' := t'}\}\{\overline{\tau'' := t''}\}\{\theta\}\\ &= \tau\{\overline{\tau'' := t''}\}\{\theta\}\\ &= \tau\{\overline{\tau'' := t''}\}\{\overline{\tau' := t'\{\overline{\tau'' := t''}\}}\}\{\theta\}\\ &= \tau(\theta)\{\overline{\tau'' := t''}\}\{\tau' := t'\{\overline{\tau'' := t''}\}\}\end{aligned}$$

- The rest of cases are straight forward.

LEMMA 15 (TYPE SUBSTITUTION). *If* $\Gamma \overset{A}{\vdash} e : t$ *then* $\Gamma\{\overline{\tau := t'}\} \overset{A}{\vdash} e : t\{\overline{\tau := t'}\}$.

PROOF. By induction over the derivation of $\Gamma \overset{A}{\vdash} e : t$.

- $\Gamma \overset{A}{\vdash} x : t$ when $\Gamma(x) = \sigma^A$ and $\sigma \succ t$
  This implies that $\sigma = \forall\overline{\tau'}.t_1$ and that there are $\overline{t_2}$ such that $t = t_1\{\overline{\tau' := t_2}\}$. Assuming $\{\overline{\tau}\} \cap \{\overline{\tau'}\} = \emptyset$ and $\mathrm{FTV}(\overline{t'}) \cap \{\overline{\tau'}\} = \emptyset$ then $\Gamma\{\overline{\tau := t'}\}(x) = \forall\overline{\tau'}.(t_1\{\overline{\tau := t'}\})^A$. As $t = t_1\{\overline{\tau' := t_2}\}$ then by lemma 14, $t\{\overline{\tau := t'}\} = t_1\{\overline{\tau' := t_2}\}\{\overline{\tau := t'}\} = t_1\{\overline{\tau := t'}\}\{\overline{\tau' := t_3}\}$ for some $\overline{t_3}$. This implies that $\sigma\{\overline{\tau := t'}\} \succ t\{\overline{\tau := t'}\}$, so:

$$\frac{}{\Gamma\{\overline{\tau := t'}\} \overset{A}{\vdash} x : t\{\overline{\tau := t'}\}}$$

- $\Gamma \overset{A}{\vdash} \lambda x.e : t_1 \to t_2$
  As

$$\frac{\Gamma, x : \forall().t_1^A \overset{A}{\vdash} e : t_2}{\Gamma \overset{A}{\vdash} \lambda x.e : t_1 \to t_2}$$

  then by the induction hypothesis $(\Gamma, x : \forall().t_1^A)\{\overline{\tau := t'}\} \overset{A}{\vdash} e : t_2\{\overline{\tau := t'}\}$ thus:

$$\frac{\Gamma, x : \forall().(t_1\{\overline{\tau := t'}\})^A \overset{A}{\vdash} e : t_2\{\overline{\tau := t'}\}}{\Gamma\overline{\tau := t'}\{\overset{\}}{\vdash}_A \lambda x.e : (t_1 \to t_2)\{\overline{\tau := t'}\}}$$

- $\Gamma \overset{A}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2$
  As

$$\frac{\Gamma \overset{A}{\vdash} e_1 : t_1 \quad \Gamma, x : \mathrm{close}(t_1, \Gamma) \overset{A}{\vdash} e_2 : t_2}{\Gamma \overset{A}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}$$

  then by the induction hypothesis $\Gamma\overline{\tau := t'}\{\overset{\}}{\vdash}_A e_1 : t_1\{\overline{\tau := t'}\}$ and $\Gamma, x : \mathrm{close}(t_1, \Gamma)\{\overline{\tau := t'}\} \overset{A}{\vdash} e_2 : t_2\{\overline{\tau := t'}\}$. As $\mathrm{close}(t_1, \Gamma) = \forall\overline{\tau'}.t_1$ where $\{\overline{\tau'}\} \cup \mathrm{FTV}(\Gamma) = \emptyset$ then it can be assumed that $\{\overline{\tau'}\} \cup \{\overline{\tau}\} = \emptyset$ and $\{\overline{\tau'}\} \cup \mathrm{FTV}(\overline{t'}) = \emptyset$. Then $\mathrm{close}(t_1, \Gamma)\{\overline{\tau := t'}\} = t''$ where $t'' = \mathrm{close}(t_1\{\overline{\tau := t'}\}, \Gamma\{\overline{\tau := t'}\})$. Then

$$\frac{\begin{array}{c}\Gamma\{\overline{\tau := t'}\} \overset{A}{\vdash} e_1 : t_1\{\overline{\tau := t'}\}\\ \Gamma\{\overline{\tau := t'}\}, x : t'' \overset{A}{\vdash} e_2 : t_2\{\overline{\tau := t'}\}\end{array}}{\Gamma\{\overline{\tau := t'}\} \overset{A}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2\{\overline{\tau := t'}\}}$$

- The remaining cases are not interesting.

LEMMA 16 (α-SUBSTITUTION). *If* $\Gamma \overset{A}{\vdash} e : t$ *then* $\Gamma\{\theta\} \overset{A\{\theta\}}{\vdash} e[\theta] : t\{\theta\}$

PROOF. Induction over derivation of $\Gamma \overset{A}{\vdash} e : t$.

LEMMA 17 (WEAKENING). *If* $\Gamma_1, \Gamma_3 \overset{A}{\vdash} e : t$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \overset{A}{\vdash} e : t$.

PROOF. Induction over the derivation of $\Gamma_1 \overset{A}{\vdash} e : t$.

LEMMA 18 (SUBSTITUTION).

$$\begin{array}{l}\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \overset{A}{\vdash} e_2 : t_2 \quad \textit{and} \\ \Gamma_1 \vdash e_1 : t_1 \quad \textit{and} \quad \{\overline{\tau}\} \cup \mathrm{FTV}(\Gamma_1) = \emptyset \\ \Longrightarrow \\ \Gamma_1, \Gamma_2 \overset{A}{\vdash} e_2[x := e_1] : t_2\end{array}$$

PROOF. By induction over the derivation of $\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \overset{A}{\vdash} e_2 : t_2$.

- $\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \vdash x : t_2$ where $\forall\overline{\tau}.t_1 \succ t_2$
  As $\forall\overline{\tau}.t_1 \succ t_2$ there is $\overline{t_3}$ such that $t_1\{\overline{\tau := t_3}\} = t_2$. By Lemma 15 $\Gamma_1\{\overline{\tau := t_3}\} \vdash e_1 : t_1\{\overline{\tau := t_3}\}$, thus $\Gamma_1 \vdash e_1 : t_2$. By Lemma 17 $\Gamma_1, \Gamma_2 \vdash e_1 : t_2$.
- $\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \vdash y : t_2$ where $\forall\overline{\tau}.t_1 \succ t_2$ and $x \neq y$
  As $(\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2)(y) = t_2 = (\Gamma_1, \Gamma_2)(y)$ then $\Gamma_1, \Gamma_2 \vdash y : t_2$.
- $\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2 \overset{A}{\vdash} \lambda y.e_2 : t_2 \to t_3$
  As
  $$\frac{\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2, y : \forall.t_2^A \overset{A}{\vdash} e_2 : t_3}{\Gamma_1, x : \forall\overline{\tau}.t_1, \Gamma_2, \overset{A}{\vdash} \lambda y.e_2 : t_2 \to t_3}$$
  then by the induction hypothesis, $\Gamma_1, \Gamma_2, y : \forall.t_2^A \overset{A}{\vdash} e_2[x := e_1] : t_3$, then:
  $$\frac{\Gamma_1, \Gamma_2, y : \forall.t_2^A \overset{A}{\vdash} e_2[x := e_1] : t_3}{\Gamma_1, \Gamma_2 \overset{A}{\vdash} (\lambda y.e_2)[x := e_1]}$$
  by assuming $x \neq y$ and $y \notin \mathrm{FV}(e_1)$.
- The rest of the cases follow this pattern

LEMMA 19 (DEMOTION).

$$\begin{array}{l}\Gamma', \alpha, \Gamma^{A,\alpha} \overset{A,\alpha,A'}{\vdash} v^i : t \\ \Longrightarrow \\ \Gamma', \alpha, \Gamma^{A} \overset{A,A'}{\vdash} v^i \downarrow^X_{\alpha,A'} : t\end{array}$$

where $i = |\alpha, A'|$ *and* $X = |\Gamma^\alpha|$.

PROOF. By induction over the derivation of $\Gamma', \alpha, \Gamma^{A,\alpha} \overset{A,\alpha,A'}{\vdash} v^i : t$.

COROLLARY 6 (READABLE DEMOTION).

$$\begin{array}{l}\Gamma, \alpha \overset{A,\alpha}{\vdash} v^1 : t \\ \Longrightarrow \\ \Gamma, \alpha \overset{A}{\vdash} v^1 \downarrow_\alpha : t\end{array}$$

PROOF. Proof: Instance of Lemma 19. □

This theorem is a copy of Theorem 4 in section 3.

THEOREM 7 (SUBJECT REDUCTION).

$$\begin{array}{l}\Gamma \overset{A}{\vdash} e : t \quad \textit{and} \quad e \longrightarrow e' \\ \Longrightarrow \\ \Gamma \overset{A}{\vdash} e' : t\end{array}$$

PROOF. By case analysis of $e \longrightarrow e'$

- $(\lambda x.e_1^0)e_2^0 \longrightarrow e_1^0[x := e_2^0]$
  As
  $$\frac{\dfrac{\Gamma, x : \forall.t'^A \overset{A}{\vdash} e_1^0 : t}{\Gamma \overset{A}{\vdash} \lambda x.e_1^0 : t' \to t} \quad \Gamma \overset{A}{\vdash} e_2^0 : t'}{\Gamma \overset{A}{\vdash} (\lambda x.e_1^0)e_2^0 : t}$$
  then by Lemma 6 $\Gamma \overset{A}{\vdash} e_1^0[x := e_2^0]$.
- $((\alpha)v^0)[\beta] \longrightarrow v^0[\alpha := \beta]$
  As
  $$\frac{\dfrac{\Gamma, \alpha \overset{A}{\vdash} e : t}{\Gamma \overset{A}{\vdash} (\alpha)e : (\alpha)t}\ \alpha \notin \Gamma}{\Gamma \overset{A}{\vdash} ((\alpha).e)[\beta] : t\{\alpha := \beta\}}$$
  and $\Gamma\{\alpha := \beta\} = \Gamma$ then by Lemma 16 $\Gamma \overset{A}{\vdash} e[\alpha := \beta] : t\{\alpha := \beta\}$.
- $\tilde{}\langle e^0 \rangle^\alpha \longrightarrow e^0$
  Follows immediately from:
  $$\frac{\dfrac{\Gamma \overset{A,\alpha}{\vdash} e : t}{\Gamma \overset{A}{\vdash} \langle e \rangle^\alpha : \langle t \rangle^\alpha}}{\Gamma \overset{A,\alpha}{\vdash} \tilde{}\langle e \rangle^\alpha : t}$$
- $\mathsf{run}\ (\alpha)\langle e^0 \rangle^\alpha \longrightarrow (\alpha)(e^0 \downarrow_\alpha)$
  $$\frac{\dfrac{\dfrac{\Gamma, \alpha \overset{A,\alpha}{\vdash} e^0 : t}{\Gamma, \alpha \overset{A}{\vdash} \langle e^0 \rangle^\alpha : \langle t \rangle^\alpha}}{\Gamma \overset{A}{\vdash} (\alpha)\langle e^0 \rangle^\alpha : (\alpha)\langle t \rangle^\alpha}}{\Gamma \overset{A}{\vdash} \mathsf{run}\ (\alpha)\langle e^0 \rangle^\alpha : (\alpha)t}$$

Note that $e_0 \in V_1$ then by corollary 7, $\Gamma, \alpha \overset{A}{\vdash} e^0 \downarrow_\alpha : t$. Then

$$\frac{\Gamma, \alpha \overset{A}{\vdash} e^0 \downarrow_\alpha : t}{\Gamma \overset{A}{\vdash} (\alpha)(e^0 \downarrow_\alpha) : (\alpha)t}$$

- $\mathsf{let}\ x = e_1^0\ \mathsf{in}\ e_2^0 \longrightarrow e_2^0[x := e_1^0]$ As

$$\frac{\Gamma \overset{A}{\vdash} e_1^0 : t' \quad \Gamma, x : \mathrm{close}(t', \Gamma) \overset{A}{\vdash} e_2^0 : t}{\Gamma \overset{A}{\vdash} \mathsf{let}\ x = e_1^0\ \mathsf{in}\ e_2^0 : t}$$

and $\mathrm{close}(t', \Gamma) = \forall \overline{\tau}.t_1$ then by Lemma 6, $\Gamma \overset{A}{\vdash} e_2^0[x := e_1^0] : t$. □