ML-like Inference for Classifiers (October 24, 2003)

Cristiano Calcagno[1], Eugenio Moggi[2⋆], and Walid Taha[3⋆⋆]

[1] Imperial College, London, UK (ccris@doc.ic.ac.uk)
[2] DISI, Univ. of Genova, v Dodecaneso 35, Genova, Italy (moggi@disi.unige.it)
[3] Rice University, TX, USA (taha@cs.rice.edu)

Abstract. Environment classifiers were recently proposed as a new approach to typing multi-stage languages. Safety was established in the simply-typed and let-polymorphic settings. While the motivation for the classifier approach was the feasibility of inference, this was in fact not established. This paper starts with the observation that inference for the full classifier-based system fails. We then identify a subset of the original system for which inference is possible. This subset, which uses *implicit classifiers*, retains significant expressivity (e.g. it can embed the calculi of Davies and Pfenning) and eliminates the need for classifier names in terms. Implicit classifiers were implemented in MetaOCaml, and no changes were needed to make an existing test suite acceptable by the new type checker.

1 Introduction

Introducing explicit staging constructs into programming languages is the goal of research projects including 'C [11, 12], Popcorn [30], MetaML [37, 23], MetaOCaml [5, 22], and Template Haskell [28]. Staging is an essential ingredient of macros [13], partial evaluation [6, 18], program generation [19], and run-time code generation [15]. In the untyped setting, the behavior of staging constructs resembles the quasi-quotation mechanisms of LISP and Scheme [3]. But in the statically-typed setting, such quotation mechanisms may prohibit static type-checking of the quoted expression. Some language designs, such as that of 'C, consider this acceptable. In Template Haskell, this is considered a feature; namely, a form of staged type inference [29]. But in the design of MetaML and MetaOCaml, it is seen as a departure from the commitment of ML and OCaml to static prevention of runtime errors.[4]

1.1 Multi-stage Basics

The use of staging constructs can be illustrated in a multi-stage language such as MetaOCaml [22] with a classic example[5]:

```
let rec power n  x = (* : int -> int code -> int code *)
  if n=0 then .<1>. else else .<.~x * .~(power (n-1) x)>.
let power72 : int -> int = .! .<fun x -> .~(power 72 .<x>.)>.
```

Ignoring the type constructor t `code` 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 staging, however, the last step just

⋆ Supported by MIUR project NAPOLI, EU project DART IST-2001-33477 and thematic network APPSEM II IST-2001-38957
⋆⋆ Supported by NSF ITR-0113569 and NSF CCR-0205542.

[4] Dynamic typing can always be introduced as an orthogonal and non-pervasive feature [1, 2, 35].

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

produces a closure that invokes the power function every time 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 enclosing bracketed computation. The application e `.<x>.` has to be performed first, 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 application of `power` is escaped to make sure that it is performed immediately. The run construct (`.!`) on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.

1.2 Background

Starting with the earliest statically typed languages supporting staging (including those of Gomard and Jones [14] and Nielson and Nielson [26]), most proposals to date fall under two distinct approaches: one treating code as always open, the other treating code as always closed. The two approaches are best exemplified by two type systems corresponding to well-known logics:

$\lambda^{\bigcirc}$ Motivated by the next modality $\bigcirc$ of linear time temporal logic, this system provides a sound framework for typing constructs that have the same operational semantics as bracket and escape [8]. As illustrated above, brackets and escapes can be used to annotate λ-abstractions so as to force evaluation under lambda. This type system supports code generation but does not provide a construct for code execution.

$\lambda^{\Box}$ Motivated by the necessity modality $\Box$ of S4 modal logic, this system provides constructs for generating and executing closed code [9]. The exact correspondence between the constructs of $\lambda^{\Box}$ and LISP-style quotation mechanism is less immediate than for $\lambda^{\bigcirc}$.

Combining the two approaches to realize a language that allows evaluation under lambda *and* a run construct is challenging [31]. In particular, evaluation under lambda gives rise to code fragments that contain free variables that are not yet "linked" to any fixed value. Running such open code fragments can produce a runtime error. Several type systems [24, 4, 36, 25] have been proposed for safely combining the key features of $\lambda^{\bigcirc}$ (the ability to manipulate open code) and $\lambda^{\Box}$ (the ability to execute closed code). But a practical solution to the problem requires meeting a number of demanding criteria simultaneously:

- **Safety:** the extension should retain static safety;
- **Conservativity:** the extension should not affect programs that do not use multi-stage facilities;
- **Inference:** the extension should support type inference;
- **Light annotations:** the extension should minimize the amount of programmer annotations required to make type inference possible.

All the above proposals were primarily concerned with the safety criterion, and were rarely able to address the others. Because previous proposals seemed notationally heavy, implementations of multi-stage languages (such MetaML and MetaOCaml) often chose to sacrifice safety. For example, in MetaOCaml `.! e` raises an exception, when the evaluation of `e` produces open code.

The type system for environment classifiers λ^{α} of [36] appears to be the most promising starting point towards fulfilling all criteria. The key feature of λ^{α} is providing a code type $\langle\tau\rangle^{\alpha}$ decorated with a classifier α that constrains the unresolved variables that may occur free in code. Intuitively, in the type system of λ^{α}, variables are declared at levels annotated by classifiers, and code of type $\langle\tau\rangle^{\alpha}$ may contain only unresolved variables declared at a level annotated with α. Classifiers are also used explicitly in terms. Type safety for λ^{α} was established [36], but type inference was only conjectured.

1.3 Contributions and Organization of this Paper

The starting point for this work is the observation that inference for full λ^α fails. To address this problem, a subset of the original system is identified for which inference is not only possible but is in fact *easy*. This subset uses *implicit classifiers*, i.e. , eliminates the need for classifier names in terms, and retains significant expressivity (e.g. , it embeds the paradigmatic calculi $\lambda^{\bigcirc}$ and $\lambda^{\Box}$). Implicit classifiers have been implemented in MetaOCaml, and no changes were needed to make an existing test suite acceptable by the type checker. The paper proceeds as follows:

- Section 2 extends a core subset of ML with environment classifiers, and proves type safety (in the sense that well typed terms cannot lead to runtime errors). The new calculus, called λ^i_{let}, corresponds to a proper subset of λ^α but eliminates classifier names in terms. This is an improvement on λ^α in making annotations lighter. Moreover, the proof of type safety for λ^α adapts easily to λ^i_{let}.
- Section 3 gives two inference algorithms:
  1. a principal typing algorithm for λ^i, the simply-typed subset of λ^i_{let} (i.e. , no type schema and let-binding), which extends Hindley's principal typing algorithm for the λ-calculus.
  2. a principal type algorithm for λ^i_{let}, which extends Damas and Milner's algorithm W.

  These results indicate that classifiers are a natural extension to well-established type systems.
- Section 4 relates λ^i to λ^α and exhibits some terms typable in λ^α that fail to have a principal type (thus, λ^α fails to meet the inference criterion). It also shows that λ^i retains significant expressivity, namely there are *typability-preserving* embeddings of $\lambda^{\bigcirc}$ and a variant of $\lambda^{\Box}$ into λ^i (similar to the embeddings into λ^α given in [36]). However, if one restricts λ^i_{let} further, by considering a runClosed construct similar to Haskell's runST [20, 21], then the embedding of $\lambda^{\Box}$ is lost (but term annotations disappear completely).
- Section 5 reports on our preliminary experience in extending the MetaOCaml implementation with the type inference algorithm for λ^i_{let}.
- Section 6 concludes and discusses further work.

Details of selected proofs as well as auxiliary definitions are included in the appendix.

1.4 Notation

Throughout the paper we use the following notation and conventions:

- We write m to range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set of its predecessors $\{i \in \mathsf{N} | i < m\}$.
- We write $\overline{a}$ to range over the set A^* of finite sequences $(a_i | i \in m)$ with $a_i \in \mathsf{A}$, and $|\overline{a}|$ denotes its length m. We write $\overline{a}_1, \overline{a}_2$ to denote the concatenation of $\overline{a}_1$ and $\overline{a}_2$.
- We write $f : A \xrightarrow{fin} B$ to say that f is a partial function from A to B with a finite domain, written $\mathrm{dom}(f)$. We write $A \to B$ to denote the set of total functions from A to B. We use the following operations on (possibly) partial functions:
  - $\{a_i : b_i | i \in m\}$ is the partial function mapping a_i to b_i (where the a_i are distinct, i.e. , $a_i = a_j$ implies $i = j$); in particular, $\emptyset$ is the everywhere undefined partial function;
  - $f \backslash a$ denotes the partial function g s.t. $g(a') = b$ iff $f(a') = b$ when $a' \neq a$, and undefined otherwise;
  - $f\{a : b\}$ denotes the (possibly) partial function g s.t. $g(a) = b$ and $g(a') = f(a')$ when $a' \neq a$;
  - f, g denotes the union of two partial functions with disjoint domains.
  - $f = g \bmod X$ means that $\forall x \in X. f(x) = g(x)$.
- We write $X \# X'$ to mean that the sets X and X' are disjoint.

Variables	$x \in \mathsf{X}$
Classifiers	$\alpha \in \mathsf{A}$
Named Levels	$A \in \mathsf{A}^*$
Terms	$e \in \mathsf{E} ::= x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e \mid \%e \mid \mathsf{open}\ e \mid \mathsf{close}\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
Type Variables	$\beta \in \mathsf{B}$
Types	$\tau \in \mathsf{T} ::= \beta \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle^{\alpha} \mid \langle \tau \rangle$
Type Schema	$\sigma \in \mathsf{S} ::= \tau \mid \forall \alpha.\sigma \mid \forall \beta.\sigma$ (equivalently $\forall \overline{\kappa}.\tau$ with $\overline{\kappa}$ sequence of distinct α and β)
Assignments	$\Gamma \in \mathsf{X} \xrightarrow{fin} (\mathsf{T} \times \mathsf{A}^*)$ of types and named levels
Assignments	$\Delta \in \mathsf{X} \xrightarrow{fin} (\mathsf{S} \times \mathsf{A}^*)$ of type schema and named levels

Fig. 1. Syntax of λ^i_{let}

2 λ^i_{let}: a Calculus with Implicit Classifiers

This section defines λ^i_{let}, an extension of the functional subset of ML with environment classifiers. In comparison to λ^α [36], classifier names do not appear in terms. In particular, the constructs of λ^α for explicit abstraction $(\alpha)\tau$ and instantiation $\tau[\alpha]$ of classifiers are replaced by the constructs close e and open e, but with more restrictive typing rules. As we show in this paper, this makes it possible to support ML-style inference of types and classifiers in a straightforward manner.

Figure 1 gives the syntax of λ^i_{let}. Intuitively, *classifiers* allow us to name parts of the environment in which a term is typed. Classifiers are described as *implicit* in λ^i_{let} because they do not appear in terms. *Named levels* are sequences of environment classifiers. They are used to keep track of the environments used as we build nested code. Named levels are thus an enrichment of the traditional notion of levels in multi-stage languages [8, 37, 33], the latter being a natural number which keeps track only of the depth of nesting of brackets. Terms include:

- the standard λ-terms, i.e. , variables drawn from an infinite set, λ-abstraction and application;
- the staging constructs of MetaML [37], i.e. , Brackets $\langle e \rangle$, escape $\tilde{}e$, and run run e, and an explicit construct $\%e$ for cross-stage persistence (CSP) [4, 36];
- the constructs close e and open e are the implicit versions of the λ^α constructs for classifiers abstraction $(\alpha)\tau$ and instantiation $\tau[\alpha]$;
- the standard let-binding for supporting Hindley-Milner polymorphism.

Types include type variables, functional types, and code types $\langle \tau \rangle^\alpha$ annotated with a classifier (exactly as in λ^α of [36], thus refining open code types). The last type $\langle \tau \rangle$ is for executable code (it is used for typing run e) and basically corresponds to the type $(\alpha)\langle \tau \rangle^\alpha$ of λ^α (as explained in more detail in Section 4.1).

As in other Hindley-Milner type systems, type schema restrict quantification at the outermost level of types. Since the types of λ^i_{let} may contain not only type variables but also classifiers, type schema allow quantification on both.

Note 1. We summarize syntax-related auxiliary definitions used in this or subsequent sections.

- $\mathrm{FV}(\_)$ denotes the set of variables free in $\_$. In λ^i_{let}, there are three kinds of variables: term variables x, classifiers α, and type variables β. The definition of $\mathrm{FV}(\_)$ for terms, types, and type schema is standard, and it extends in the obvious way to A, Γ, and Δ, e.g. , $\mathrm{FV}(\Delta) = \cup\{\mathrm{FV}(\sigma) \cup \mathrm{FV}(A) \mid \Delta(x) = \sigma^A\}$.

$$\text{var}\ \frac{\sigma \succ \tau}{\Delta \overset{A}{\vdash} x : \tau}\ \Delta(x) = \sigma^A \qquad \text{lam}\ \frac{\Delta\{x : \tau_1^A\} \overset{A}{\vdash} e : \tau_2}{\Delta \overset{A}{\vdash} \lambda x.e : \tau_1 \to \tau_2} \qquad \text{app}\ \frac{\Delta \overset{A}{\vdash} e_1 : \tau_1 \to \tau_2 \quad \Delta \overset{A}{\vdash} e_2 : \tau_1}{\Delta \overset{A}{\vdash} e_1 e_2 : \tau_2} \qquad \text{brck}\ \frac{\Delta \overset{A,\alpha}{\vdash} e : \tau}{\Delta \overset{A}{\vdash} \langle e \rangle : \langle \tau \rangle^\alpha}$$

$$\text{esc}\ \frac{\Delta \overset{A}{\vdash} e : \langle \tau \rangle^\alpha}{\Delta \overset{A,\alpha}{\vdash} \tilde{}e : \tau} \qquad \text{run}\ \frac{\Delta \overset{A}{\vdash} e : \langle \tau \rangle}{\Delta \overset{A}{\vdash} \mathsf{run}\ e : \tau} \qquad \text{csp}\ \frac{\Delta \overset{A}{\vdash} e : \tau}{\Delta \overset{A,\alpha}{\vdash} \%e : \tau} \qquad \text{open}\ \frac{\Delta \overset{A}{\vdash} e : \langle \tau \rangle}{\Delta \overset{A}{\vdash} \mathsf{open}\ e : \langle \tau \rangle^\alpha}$$

$$\text{close}\ \frac{\Delta \overset{A}{\vdash} e : \langle \tau \rangle^\alpha}{\Delta \overset{A}{\vdash} \mathsf{close}\ e : \langle \tau \rangle}\ \alpha \notin \mathrm{FV}(\Delta, A, \tau) \qquad \text{let}\ \frac{\Delta \overset{A}{\vdash} e_1 : \tau_1 \quad \Delta\{x : (\forall \overline{\kappa}.\tau_1)^A\} \overset{A}{\vdash} e_2 : \tau_2}{\Delta \overset{A}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \mathrm{FV}(\Delta, A) \# \overline{\kappa}$$

Fig. 2. Type System for λ^i_{let}

- We write $\equiv$ for equivalence up to α-conversion on terms, types, and type schema. $\_[x : e]$ denotes substitution of x with e in $\_$ modulo $\equiv$, i.e. , the bound variables in $\_$ are automatically renamed to avoid clashes with $\mathrm{FV}(e)$. Similarly we write $\_[\alpha : \alpha']$ and $\_[\beta : \tau]$ for classifiers and type variables.
- We write Sub for the set of *substitutions*, i.e. , functions (with domain $\mathsf{A} \cup \mathsf{B}$) mapping classifiers α to classifiers and type variables β to types τ, and having a finite *support* $\{\alpha | \rho(\alpha) \neq \alpha\} \cup \{\beta | \rho(\beta) \neq \beta\}$. Then $\_[\rho]$ denotes *parallel substitution*, where each free occurrence in $\_$ of a classifier α and term variable β is replaced by its ρ-image. With some abuse of notation, we write $e[\rho]$ also when ρ is a partial function with finite domain, by extending it as the identity outside $\mathrm{dom}(\rho)$.
- $\sigma \succ \tau$ means that type τ is an *instance* of σ, i.e. ,
  - $\forall \overline{\kappa}.\tau \succ \tau' \overset{\Delta}{\Longleftrightarrow} \tau[\rho] = \tau'$ for some $\rho \in \mathsf{Sub}$ with support $\overline{\kappa}$ (the order in $\overline{\kappa}$ is irrelevant)

  and $\succ$ extends to a relation on type schemas and type schema assignments:
  - $\forall \overline{\kappa}.\tau \succ \forall \overline{\kappa}'.\tau' \overset{\Delta}{\Longleftrightarrow} \forall \overline{\kappa}.\tau \succ \tau'$, we can assume $\overline{\kappa}' \# \mathrm{FV}(\forall \overline{\kappa}.\tau)$ by α-conversion
  - $\Delta_1 \succ \Delta_2 \overset{\Delta}{\Longleftrightarrow} \mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$ and $\sigma_1 \succ \sigma_2$ and $A_1 = A_2$ whenever $\sigma_1^{A_1} = \Delta_1(x)$ and $\sigma_2^{A_2} = \Delta_2(x)$.

2.1 Type System

Figure 2 gives the type system for λ^i_{let}. The first three rules are mostly standard. As in ML, a polymorphic variable x whose type schema is σ can be assigned any type which is an instance of σ. 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-checked 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.

The rule for brackets is almost the same as in previous type systems. First, for every code type a classifier must be assigned. 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 is used in both the variable and the escape rules to make sure that only variables and code fragments with the same classification are ever incorporated into this code fragment. The escape rule at named level A, α only allows the incorporation of code fragments of type $\langle \tau \rangle^\alpha$. The rule for CSP itself is standard: It allows us to incorporate a term e at a "higher" level. The rule for run allows to execute a code fragment that has type $\langle \tau \rangle$.

The next two rules are introduction and elimination for the runnable code type $\langle\tau\rangle$. One rule says that $\mathsf{close}\ e$ is runnable code when e can be classified with *any* α, conversely the other rule says that code $\mathsf{open}\ e$ can be classified by any α provided e is runnable code. The rule for let is standard and allows the introduction of variables of polymorphic type.

The following proposition summarizes the key properties of the type system relevant for type safety as well as type inference.

Proposition 1 (TS properties). *The following rules are admissible:*

$$-\ \alpha\tau\text{-subst}\ \frac{\Delta \overset{A}{\vdash} e : \tau}{(\Delta \overset{A}{\vdash} e : \tau)[\rho]}\ \rho \in \mathsf{Sub} \qquad \Delta\text{-sub}\ \frac{\Delta_2 \overset{A}{\vdash} e : \tau}{\Delta_1 \overset{A}{\vdash} e : \tau}\ \Delta_1 \succ \Delta_2$$

$$-\ \text{strength}\ \frac{\Delta \overset{A}{\vdash} e : \tau}{\Delta \backslash x \overset{A}{\vdash} e : \tau}\ x \notin \mathrm{FV}(e) \qquad \text{weaken}\ \frac{\Delta \overset{A}{\vdash} e : \tau}{\Delta, x : \sigma_1^{A_1} \overset{A}{\vdash} e : \tau}\ x \notin \mathit{dom}(\Delta)$$

$$-\ e\text{-subst}\ \frac{\Delta \overset{A_1}{\vdash} e_1 : \tau_1 \qquad \Delta, x : (\forall \overline{\kappa}.\tau_1)^{A_1} \overset{A_2}{\vdash} e_2 : \tau_2}{\Delta \overset{A_2}{\vdash} e_2[x : e_1] : \tau_2}\ \overline{\kappa} \# \mathrm{FV}(\Delta, A_1)$$

Proof. Interesting cases in the proofs of these properties are outlined in Appendix A.

2.2 Operational Semantics and Type Safety

Figure 3 gives the evaluation rules of the big-step operational semantics for deriving judgments of the form $e \overset{n}{\hookrightarrow} v$. This function is essentially the same as that for λ^α. For establishing type safety (of a big-step operational semantics), one must spell out when evaluation causes an error. This is done in Figure 8, by giving rules deriving judgments of the form $e \overset{n}{\hookrightarrow} \mathsf{err}$.

Auxiliary definitions, like Demotion, and technical lemmas are similar but also simpler than those for λ^α, because of the absence of classifier names in terms.

Lemma 1 (Promotion and Demotion). *The following rules are admissible:*

$$\text{promotion}\ \frac{\Delta \overset{A}{\vdash} e : \tau}{\Delta \overset{\alpha,A}{\vdash} \%_{|A|} e : \tau} \qquad \text{demotion}\ \frac{\Delta_1, \Delta_2^\alpha \overset{\alpha,A}{\vdash} v : \tau \quad v \in \mathsf{V}^{|A|+1}}{\Delta_1, \Delta_2 \overset{A}{\vdash} v \downarrow_{|A|}^{|\Delta_2|} : \tau}\ \alpha \notin \mathrm{FV}(\Delta_1)$$

Proposition 2 (SOS properties). *Let Δ^+ be any Δ such that $\Delta(x) \neq \sigma^\emptyset$ for any x, then*

1. *$e \overset{n}{\hookrightarrow} e'$ implies $e' \in \mathsf{V}^n$.*
2. *$\Delta^+ \overset{A}{\vdash} e : \tau$ and $e \overset{|A|}{\hookrightarrow} e'$ imply $\Delta^+ \overset{A}{\vdash} e' : \tau$.*
3. *$\Delta^+ \overset{A}{\vdash} e : \tau$ implies not $e \overset{|A|}{\hookrightarrow} \mathsf{err}$*

As a corollary, we get that well-typed programs cannot cause a run-time error.

Theorem 1 (Type Safety). *If $\emptyset \overset{\emptyset}{\vdash} e : \tau$, then $e \overset{0}{\hookrightarrow} \mathsf{err}$ is not derivable.*

$$
\begin{array}{ll}
\text{Levels} & n \in \mathsf{N} ::= 0 \mid n+ \\
\text{Assignments} & \gamma \in \mathsf{X} \xrightarrow{fin} \mathsf{N} \text{ of levels} \\
\text{Values} & v^0 \in \mathsf{V}^0 ::= \lambda x.e \mid \langle v^1 \rangle \mid \mathsf{close}\ v^0 \\
& v^{n+} \in \mathsf{V}^{n+} ::= x \mid \lambda x.v^{n+} \mid v_1^{n+} v_2^{n+} \mid \langle v^{n++} \rangle \mid \%v^n \mid \mathsf{run}\ v^{n+} \mid \mathsf{close}\ v^{n+} \mid \mathsf{open}\ v^{n+} \mid \\
& \qquad \mathsf{let}\ x = v_1^{n+}\ \mathsf{in}\ v_2^{n+} \\
& v^{n++} \in \mathsf{V}^{n++} + = \tilde{}\, v^{n+} \qquad \text{(in addition to productions of BNF for } \mathsf{V}^{n+}\text{)}
\end{array}
$$

Normal evaluation rules:

$$
x \overset{n+}{\hookrightarrow} x \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v \quad e[x:v] \overset{0}{\hookrightarrow} v'}{e_1 e_2 \overset{0}{\hookrightarrow} v'} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{e_1 e_2 \overset{n+}{\hookrightarrow} v_1 v_2} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v \rangle \quad v \downarrow_0^{\emptyset} \overset{0}{\hookrightarrow} v'}{\mathsf{run}\ e \overset{0}{\hookrightarrow} v'}
$$

$$
\frac{e \overset{n+}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\tilde{}\, e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\tilde{}\, e \overset{n++}{\hookrightarrow} \tilde{}\, v} \qquad \frac{e \overset{n}{\hookrightarrow} v}{\%e \overset{n+1}{\hookrightarrow} \%v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{run}\ e \overset{n+}{\hookrightarrow} \mathsf{run}\ v} \qquad \frac{e \overset{n}{\hookrightarrow} v}{\mathsf{close}\ e \overset{n}{\hookrightarrow} \mathsf{close}\ v}
$$

$$
\frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ v}{\mathsf{open}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\mathsf{open}\ e \overset{n+1}{\hookrightarrow} \mathsf{open}\ v} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \quad e_2[x:v] \overset{0}{\hookrightarrow} v'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} v'} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{n+}{\hookrightarrow} \mathsf{let}\ x = v_1\ \mathsf{in}\ v_2}
$$

Demotion and Auxiliary Definitions:

$$
x \downarrow_n^\gamma \equiv x \text{ if } \gamma(x) = n \quad (\lambda x.e) \downarrow_n^\gamma \equiv \lambda x.(e \downarrow_n^{\gamma\{x:n\}}) \quad (e_1 e_2) \downarrow_n^\gamma \equiv e_1 \downarrow_n^\gamma e_2 \downarrow_n^\gamma
$$

$$
\langle e \rangle \downarrow_n^\gamma \equiv \langle e \downarrow_{n+}^\gamma \rangle \quad (\tilde{}\, e) \downarrow_{n+}^\gamma \equiv \tilde{}\, (e \downarrow_n^\gamma) \quad (\mathsf{run}\ e) \downarrow_n^\gamma \equiv \mathsf{run}\ (e \downarrow_n^\gamma) \quad (\%e) \downarrow_{n+}^\gamma \equiv \%(e \downarrow_n^\gamma) \quad (\%e) \downarrow_0^\gamma \equiv e[\%\gamma]
$$

$$
(\mathsf{close}\ e) \downarrow_n^\gamma \equiv \mathsf{close}\ (e \downarrow_n^\gamma) \quad \mathsf{open}\ e \downarrow_n^\gamma \equiv \mathsf{open}\ e \downarrow_n^\gamma \quad (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \downarrow_n^\gamma \equiv \mathsf{let}\ x = (e_1 \downarrow_n^\gamma)\ \mathsf{in}\ (e_2 \downarrow_n^{\gamma\{x:n\}})
$$

$$
\%_0 e \equiv \%e \quad \%_{n+} e \equiv \tilde{}\, (\%_n \langle e \rangle) \quad \%\gamma(x) \equiv \%_n x \text{ if } \gamma(x) = n \quad |\Delta|(x) = |A| \text{ if } \Delta(x) = \sigma^A \quad \Delta^\alpha(x) = \sigma^{\alpha, A} \text{ if } \Delta(x) = \sigma^A
$$

Fig. 3. Big-step Operational Semantics

3 Inference Algorithms

This section describes two inference algorithms. The first algorithm extends Hindley's principal typing algorithm [16] for the simply typed λ-calculus with type variables to λ^i (the simply-typed subset of λ^i_{let}, i.e. , without type schema and let-binding). Existence of principal typings is very desirable but hard to get (see [17, 38]). Thus it is reassuring that it is retained after the addition of classifiers.

The second algorithm extends Damas and Milner's algorithm W [7] to λ^i_{let} and proves that it is sound and complete for deriving principal types. Damas and Milner's algorithm is at the core of type inference for languages such as ML, OCaml, and Haskell. That this extension is possible (and easy) is of paramount importance to the practical use of the proposed type system.

Both algorithms make essential use of a function $mgu(T)$ computing a most general unifier $\rho \in \mathsf{Sub}$ for a finite set T of equations between types or between classifiers. Section B.1 in the appendix recalls the basic properties of mgu in the setting of many-sorted algebra. For convenience, we also introduce the following derived notation for sets of equations (used in some side-conditions to the rules describing the algorithms):

- (A_1, A_2) denotes $\{(\alpha_{1,i}, \alpha_{2,i}) \mid i \in n\}$ when $n = |A_1| = |A_2|$, and is undefined when $|A_1| \neq |A_2|$
- (Γ_1, Γ_2) denotes $\cup\{(\tau_{1,x}, \tau_{2,x}), (A_{1,x}, A_{2,x}) | x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)\}$ where $\Gamma_i(x) = \tau_{i,x}^{A_{i,x}}$.

3.1 Principal typing

We extend Hindley's principal typing algorithm for the simply typed λ-calculus with type variables to λ^i. Wells [38] gives a general definition of principal typing and related notions, but we need to adapt his definition of Hindley's principal typing to our setting, mainly to take into account levels.

Definition 1 (Typing). *A triple (Γ, τ, A) is a typing of (e, n) $\stackrel{\Delta}{\Longleftrightarrow}$ $\Gamma \stackrel{A}{\vdash} e : \tau$ is derivable and $n = |A|$. A Hindley principal typing of (e, n) is a typing (Γ, τ, A) of (e, n) s.t.*

- *$\Gamma' \stackrel{A'}{\vdash} e : \tau'$ with $n = |A'|$ implies $\Gamma[\rho] \subseteq \Gamma'$ and $\tau' = \tau[\rho]$ and $A' = A[\rho]$ for some $\rho \in \mathsf{Sub}$.*

Remark 1. Usually one assigns typings to terms. We have chosen to assign typings to a pair (e, n), because the operational semantics of a term is level-dependent. However, one can easily assign typings to terms (and retain the existence of principal typings). First, we introduce an infinite set of variables $\phi \in \Phi$ ranging over annotated levels. Then, we modify the BNF for annotated levels to become $A ::= \phi \mid A, \alpha$. Unification will also have to deal with equations for annotated levels, e.g. , $\phi_1 = \phi_2, \alpha$. A posteriori, one can show that a principal typing will contain exactly one variable ϕ.

Figure 4 defines the algorithm by giving a set of rules (directed by the structure of e) for deriving judgments of the form $\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A)$. $\mathcal{K} \subseteq_{fin} \mathsf{A} \uplus \mathsf{B}$ is an auxiliary parameter (instrumental to the algorithm), which is threaded in recursive calls for recording the classifiers and type variables used so far. The algorithm either computes a typing (and updates $\mathcal{K}$) or fails. The algorithm enjoys the following properties, which imply that every (e, n) with a typing has a principal typing.

Theorem 2 (Soundness). *If $\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma', \tau', A')$, then $\Gamma' \stackrel{A'}{\vdash} e : \tau'$ and $n = |A'|$, moreover $dom(\Gamma') = \mathrm{FV}(e)$, $\mathcal{K} \subseteq \mathcal{K}'$ and $\mathrm{FV}(\Gamma', \tau', A') \subseteq \mathcal{K}' \setminus \mathcal{K}$.*

Theorem 3 (Completeness). *If $\Gamma' \stackrel{A'}{\vdash} e : \tau'$, then $\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A)$ is derivable (for any choice of $\mathcal{K}$) and exists $\rho' \in \mathsf{Sub}$ s.t. $\Gamma[\rho'] \subseteq \Gamma'$, $\tau' = \tau[\rho']$ and $A' = A[\rho']$.*

Moreover, from general properties of the most general unifier and the similarity of our principal typing algorithm with that for the λ-calculus, one can also show that:

Theorem 4 (Conservative Extension). *If $e \in \mathsf{E}_\lambda$, then (Γ, τ) is a principal typing of e in λ $\Longleftrightarrow$ $(\Gamma, \tau, \emptyset)$ is a principal typing of $(e, 0)$ in λ^i, where we identify $x : \tau$ with $x : \tau^{\emptyset}$.*

3.2 Principal Type Inference

In this section, we extend Damas and Milner's [7] principal type algorithm to λ^i_{let} and prove that it is sound and complete. Also in this case we have to adapt to our setting the definition of Damas-Milner principal type in [38].

Definition 2 (Principal Type). *A Damas-Milner principal type of (Δ, A, e) is a type τ s.t.*

1. *$\Delta \stackrel{A}{\vdash} e : \tau$.*
2. *$\Delta \stackrel{A}{\vdash} e : \tau'$ implies $\tau' = \tau[\rho]$ for some $\rho \in \mathsf{Sub}$ with support $\mathrm{FV}(\tau) - \mathrm{FV}(\Delta, A)$*

$$\frac{\beta \text{ and } A = (\alpha_i | i \in n) \text{ distinct and } \notin \mathcal{K}}{\mathcal{K}, (x, n) \implies \mathcal{K} \uplus \{\beta, A\}, (x : \beta^A, \beta, A)} \qquad \frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad x \notin \mathrm{FV}(e) \text{ and } \beta \notin \mathcal{K}'}{\mathcal{K}, (\lambda x.e, n) \implies \mathcal{K}' \uplus \{\beta\}, (\Gamma, \beta \to \tau, A)}$$

$$\frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau_2, A_2) \quad x \in \mathrm{FV}(e) \text{ and } \Gamma(x) = \tau_1^{A_1} \text{ and } \rho = mgu(A_1, A_2)}{\mathcal{K}, (\lambda x.e, n) \implies \mathcal{K}' \uplus \{\beta\}, (\Gamma \setminus x, \tau_1 \to \tau_2, A_2)[\rho]}$$

$$\frac{\begin{array}{l}\mathcal{K}, (e_1, n) \implies \mathcal{K}', (\Gamma_1, \tau_1, A_1) \\ \mathcal{K}', (e_2, n) \implies \mathcal{K}'', (\Gamma_2, \tau_2, A_2) \quad \rho = mgu((\tau_1, \tau_2 \to \beta), (\Gamma_1, \Gamma_2), (A_1, A_2))\end{array}}{\mathcal{K}, (e_1\ e_2, n) \implies \mathcal{K}'' \uplus \{\beta\}, (\Gamma_1 \cup \Gamma_2, \beta, A_1)[\rho]}$$

$$\frac{\mathcal{K}, (e, n+) \implies \mathcal{K}', (\Gamma, \tau, A\alpha)}{\mathcal{K}, (\langle e \rangle, n) \implies \mathcal{K}', (\Gamma, \langle \tau \rangle^\alpha, A)} \qquad \frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad \rho = mgu(\tau, \langle \beta \rangle^\alpha) \text{ and } \beta, \alpha \notin \mathcal{K}'}{\mathcal{K}, (\tilde{}e, n+) \implies \mathcal{K}' \uplus \{\beta, \alpha\}, (\Gamma, \beta, A\alpha)[\rho]}$$

$$\frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad \alpha \notin \mathcal{K}'}{\mathcal{K}, (\%e, n+) \implies \mathcal{K}' \uplus \{\alpha\}, (\Gamma, \tau, A\alpha)} \qquad \frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad \rho = mgu(\tau, \langle \beta \rangle) \text{ and } \beta \notin \mathcal{K}'}{\mathcal{K}, (\mathsf{run}\ e, n) \implies \mathcal{K}' \uplus \{\beta\}, (\Gamma, \beta, A)[\rho]}$$

$$\frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad \rho = mgu(\tau, \langle \beta \rangle^\alpha) \text{ and } \beta, \alpha \notin \mathcal{K}'}{\mathcal{K}, (\mathsf{open}\ e, n) \implies \mathcal{K}' \uplus \{\beta, \alpha\}, (\Gamma, \langle \beta \rangle^\alpha, A)[\rho]}$$

$$\frac{\mathcal{K}, (e, n) \implies \mathcal{K}', (\Gamma, \tau, A) \quad \rho = mgu(\tau, \langle \beta \rangle^\alpha) \text{ and } \beta, \alpha \notin \mathcal{K}' \text{ and } \alpha[\rho] \notin \mathrm{FV}((\Gamma, \beta, A)[\rho])}{\mathcal{K}, (\mathsf{close}\ e, n) \implies \mathcal{K}'' \uplus \{\beta, \alpha\}, \Gamma, \langle \beta \rangle, A)[\rho]}$$

Fig. 4. Principal Typing Algorithm

We define a principal type algorithm $W(\Delta, A, e, \mathcal{K})$, where $\mathcal{K} \subseteq_{fin} \mathsf{A} \uplus \mathsf{B}$ is an auxiliary parameter that is threaded in recursive calls for recording the classifiers and type variables used so far. The algorithm either computes a type and a substitution for Δ and A (and updates $\mathcal{K}$) or fails. Figure 5 derives judgments of the form $\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau)$. When the judgment is derivable, it means that $W(\Delta, A, e, \mathcal{K}) = (\rho, \tau, \mathcal{K}')$. The rules use the following notation:

- $\mathrm{close}(\tau, \Delta, A) \stackrel{\Delta}{\equiv} \forall \overline{\kappa}.\tau$, where $\overline{\kappa} = \mathrm{FV}(\tau) - \mathrm{FV}(\Delta, A)$ (the order in $\overline{\kappa}$ is irrelevant)
- $\rho'\rho$ denotes composition of substitutions, i.e. , $e[\rho'\rho] = (e[\rho])[\rho']$

The algorithm enjoys the following soundness and completeness properties. Details of selected proofs are included in Appendix B.

Theorem 5 (Soundness). *If* $W(\Delta, A, e, \mathcal{K}) = (\rho, \tau, \mathcal{K}')$ *and* $\mathrm{FV}(\Delta, A) \subseteq \mathcal{K}$ *then* $\Delta[\rho] \overset{A[\rho]}{\vdash} e : \tau$, *moreover* $\mathcal{K} \subseteq \mathcal{K}'$ *and* $\mathrm{FV}(\tau, \Delta[\rho], A[\rho]) \subseteq \mathcal{K}'$.

Proof. The proof is by induction on the computation of $W(\Delta, A, e, \mathcal{K})$.

Theorem 6 (Completeness). *If* $\Delta[\rho'] \overset{A[\rho']}{\vdash} e : \tau'$ *and* $\mathrm{FV}(\Delta, A) \subseteq \mathcal{K} \subseteq_{fin} \mathsf{A} \cup \mathsf{B}$ *then* $W(\Delta, A, e, \mathcal{K}) = (\rho, \tau, \mathcal{K}')$ *is defined and exists* $\rho'' \in \mathsf{Sub}$ *s.t.* $\tau' = \tau[\rho'']$ *and* $\Delta[\rho'] \equiv \Delta[\rho''\rho]$ *and* $A[\rho'] = A[\rho''\rho]$.

Proof. The proof is by induction on the structure of e.

Remark 2. In practice, one is interested in typing a complete program e, i.e. , in computing the principal typing for $(\emptyset, \emptyset, e)$. If the algorithm returns a pair (ρ, τ), then τ is the principal $(\emptyset, \emptyset, e)$, and ρ can be ignored. Even when the program uses a library, one can ignore the substitution ρ, since $\mathrm{FV}(\Delta) = \emptyset$.

$$\frac{\Delta(x) \equiv (\forall \overline{\kappa}.\tau)^{A_1} \quad \rho = mgu(A, A_1) \quad \overline{\kappa} \# \mathcal{K}}{\mathcal{K}, (\Delta, x, A) \implies \mathcal{K} \uplus \{\overline{\kappa}\}, (\rho, \tau[\rho])} \qquad \frac{\mathcal{K} \uplus \{\beta\}, (\Delta\{x : \beta^A\}, e, A) \implies \mathcal{K}', (\rho, \tau) \quad \beta \notin \mathcal{K}}{\mathcal{K}, (\Delta, \lambda x.e, A) \implies \mathcal{K}', (\rho, \beta[\rho] \to \tau)}$$

$$\frac{\begin{array}{l}\mathcal{K}, (\Delta, e_1, A) \implies \mathcal{K}', (\rho_1, \tau_1) \\ \mathcal{K}', (\Delta[\rho_1], e_2, A[\rho_1]) \implies \mathcal{K}'', (\rho_2, \tau_2) \quad \rho = mgu(\tau_1[\rho_2], \tau_2 \to \beta) \quad \beta \notin \mathcal{K}''\end{array}}{\mathcal{K}, (\Delta, e_1 e_2, A) \implies \mathcal{K}'' \uplus \{\beta\}, (\rho\rho_2\rho_1, \beta[\rho])}$$

$$\frac{\mathcal{K} \uplus \{\alpha\}, (\Delta, e, (A, \alpha)) \implies \mathcal{K}', (\rho, \tau) \quad \alpha \notin \mathcal{K}}{\mathcal{K}, (\Delta, \langle e \rangle, A) \implies \mathcal{K}', (\rho, \langle \tau \rangle^{\alpha[\rho]})} \qquad \frac{\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau) \quad \rho' = mgu(\tau, \langle \beta \rangle^{\alpha}) \quad \beta \notin \mathcal{K}'}{\mathcal{K}, (\Delta, \tilde{}e, (A, \alpha)) \implies \mathcal{K}' \uplus \{\beta\}, (\rho'\rho, \beta[\rho'])}$$

$$\frac{\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau)}{\mathcal{K}, (\Delta, \%e, (A, \alpha)) \implies \mathcal{K}', (\rho, \tau)} \qquad \frac{\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau) \quad \rho' = mgu(\tau, \langle \beta \rangle) \quad \beta \notin \mathcal{K}'}{\mathcal{K}, (\Delta, \mathsf{run}\ e, A) \implies \mathcal{K}' \uplus \{\beta\}, (\rho'\rho, \beta[\rho'])}$$

$$\frac{\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau) \quad \rho' = mgu(\tau, \langle \beta \rangle) \quad \alpha, \beta \notin \mathcal{K}'}{\mathcal{K}, (\Delta, \mathsf{open}\ e, A) \implies \mathcal{K}' \uplus \{\alpha, \beta\}, (\rho'\rho, \langle \beta[\rho'] \rangle^{\alpha})}$$

$$\frac{\mathcal{K}, (\Delta, e, A) \implies \mathcal{K}', (\rho, \tau) \quad \rho' = mgu(\tau, \langle \beta \rangle^{\alpha}) \quad \alpha, \beta \notin \mathcal{K}' \quad \alpha[\rho'] \notin \mathrm{FV}(\Delta[\rho'\rho], A[\rho'\rho], \beta[\rho'])}{\mathcal{K}, (\Delta, \mathsf{close}\ e, A) \implies \mathcal{K}' \uplus \{\alpha, \beta\}, (\rho'\rho, \langle \beta[\rho'] \rangle)}$$

$$\frac{\mathcal{K}, (\Delta, e_1, A) \implies \mathcal{K}', (\rho_1, \tau_1) \quad \mathcal{K}', (\Delta[\rho_1]\{x : \mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1])^{A[\rho_1]}\}, e_2, A[\rho_1]) \implies \mathcal{K}'', (\rho_2, \tau_2)}{\mathcal{K}, (\Delta, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, A) \implies \mathcal{K}'', (\rho_2\rho_1, \tau_2)}$$

Fig. 5. Principal Type Algorithm

4 Relation to other calculi

This section studies the expressivity of the type system for λ^i, the simply-typed subset of λ^i_{let} (i.e. , no let-binding and no quantification in type schema). The typing judgment for λ^i takes the form $\Gamma \overset{A}{\vdash} e : \tau$, since type schema collapse into types, and the typing rules are restricted accordingly. In summary, we have the following results:

- λ^i is a proper subset of λ^α, but the additional expressivity of λ^α comes at a price: the type system has *no principal types*.
- λ^i retains significant expressivity, namely, the embeddings given in [36] for two paradigmatic calculi $\lambda^{\bigcirc}$ and λ^{S4} factor through λ^i.
- λ^i can be simplified further, by introducing a construct $\mathsf{runClosed}\ e$ similar to Haskell's runST, but doing so implies that the embedding of λ^{S4} no longer holds.

4.1 Relation to λ^α

The key feature of λ^α is the inclusion of a special quantifier $(\alpha)\tau$ in the language of types, representing universal quantification over classifiers. Figure 6 recalls the BNF for terms and types, and the most relevant typing rules [36]. In λ^i the main difference is that the quantifier $(\alpha)\tau$ of λ^α is replaced by the runnable code type $\langle \tau \rangle$. In fact, $\langle \tau \rangle$ corresponds to a restricted form of quantification, namely $(\alpha)\langle \tau \rangle^\alpha$ with $\alpha \notin \mathrm{FV}(\tau)$. It is difficult to define formally a typability-preserving embedding of λ^i into λ^α, since we need to recover classifier names in terms. Therefore, we justify the correspondence at the level of terms only informally:

$$\begin{array}{rl} \text{Terms} & e \in \mathsf{E} ::= x \mid \lambda x.e \mid e\ e \mid \langle e \rangle^{\alpha} \mid \tilde{}e \mid \%e \mid \mathsf{run}\ e \mid (\alpha)e \mid e[\alpha] \\ \text{Types} & \tau \in \mathsf{T} ::= \beta \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle^{\alpha} \mid (\alpha)\tau \end{array}$$

$$\text{brck}\ \frac{\Gamma \overset{A,\alpha}{\vdash} e : \tau}{\Gamma \overset{A}{\vdash} \langle e \rangle^{\alpha} : \langle \tau \rangle^{\alpha}} \qquad \text{esc}\ \frac{\Gamma \overset{A}{\vdash} e : \langle \tau \rangle^{\alpha}}{\Gamma \overset{A,\alpha}{\vdash} \tilde{}e : \tau} \qquad \text{csp}\ \frac{\Gamma \overset{A}{\vdash} e : \tau}{\Gamma \overset{A,\alpha}{\vdash} \%e : \tau} \qquad \text{all-run}\ \frac{\Gamma \overset{A}{\vdash} e : (\alpha)\langle \tau \rangle^{\alpha}}{\Gamma \overset{A}{\vdash} \mathsf{run}\ e : (\alpha)\tau}$$

$$\text{all-close}\ \frac{\Gamma \overset{A}{\vdash} e : \tau}{\Gamma \overset{A}{\vdash} (\alpha)e : (\alpha)\tau}\ \alpha \notin \mathrm{FV}(\Gamma, A) \qquad \text{all-open}\ \frac{\Gamma \overset{A}{\vdash} e : (\alpha)\tau}{\Gamma \overset{A}{\vdash} e[\alpha'] : \tau[\alpha : \alpha']}$$

Fig. 6. Type System for λ^{α} (adapted from [36])

- The terms $\mathsf{open}\ e$ and $\mathsf{close}\ e$ of λ^{i} correspond to $(\alpha)e$ and $e[\alpha]$ of λ^{α}. Since λ^{i} has no classifier names in terms, these constructs record that a classifier abstraction and instantiation has occurred *without* naming the classifier involved. (Similarly, the term $\langle e \rangle$ in λ^{i} corresponds to $\langle e \rangle^{\alpha}$ in λ^{α}.)
- The term $\%e$ has exactly the same syntax and meaning in the two calculi.
- The term $\mathsf{run}\ e$ of λ^{i} corresponds to $(\mathsf{run}\ e)[\alpha']$ of λ^{α}, where α' can be chosen arbitrarily without changing the result type. In fact, the type of run in λ^{α} is $((\alpha)\langle \tau \rangle^{\alpha}) \to (\alpha)\tau$, while in λ^{i} it is $\langle \tau \rangle \to \tau$, which corresponds to $((\alpha)\langle \tau \rangle^{\alpha}) \to \tau$ with $\alpha \notin \mathrm{FV}(\tau)$.

We conclude the comparison between λ^{i} and λ^{α} by showing that type inference in λ^{α} is problematic.

Lack of principal types in λ^{α}. Consider the closed term $e \equiv (\lambda x.\mathsf{run}\ x)$. We can assign to e exactly the types of the form $((\alpha)\langle \tau \rangle^{\alpha}) \to (\alpha)\tau$ with an arbitrary type τ, including ones with $\alpha \in \mathrm{FV}(\tau)$. However, e does not have a *principal type*, i.e. , one from which one can recover all other types (modulo α-conversion of bound classifiers) by applying a substitution $\rho \in \mathsf{Sub}$ for classifiers and type variables. In fact, the obvious candidate for the principal type, i.e. , $((\alpha)\langle \beta \rangle^{\alpha}) \to (\alpha)\beta$, allows us to recover only the types of the form $((\alpha)\langle \tau \rangle^{\alpha}) \to (\alpha)\tau$ with $\alpha \notin \mathrm{FV}(\tau)$, since substitution should be capture avoiding.

Lack of principal types in previous polymorphic extensions of λ^{α}. A more expressive type system for λ^{α} was previously proposed [36], where type variables β are replaced by variables β^{n} ranging over types parameterized w.r.t. n classifiers. Thus, the BNF for types becomes:

$$\tau \in \mathsf{T} ::= \beta^{n}[A] \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle^{\alpha} \mid (\alpha)\tau \quad \text{with } |A| = n$$

In this way, there is a better candidate for the principal type of e, namely $((\alpha)\langle \beta^{1}[\alpha] \rangle^{\alpha}) \to (\alpha)\beta^{1}[\alpha]$.

In this extension, standard unification techniques are no longer applicable, and some form of higher-order unification is needed. However, even in this system, there are typable terms that do not have a principal type. For instance, the term $e = (x(x_1[\alpha]), f(x_2[\alpha]))$ (for simplicity, we assume that we have pairing and product types) has no principal typing, in fact

- x must be a function, say of type $\tau \to \tau'$
- x_i must be of type $(\alpha_i)\tau_i$, among them the most general is $(\alpha_i)\beta_i^{1}[\alpha_i]$
- τ and $\tau_i[\alpha_i : \alpha]$ must be the same, but there is no most general unifier for $\beta_1^{1}[\alpha] = \beta_2^{1}[\alpha]$.

4.2 Embedding of $\lambda^{\bigcirc}$

The embedding of $\lambda^{\bigcirc}$ [8] into λ^{i} is direct. We pick one arbitrary classifier α and define the embedding as follows:

$$[\![\beta]\!] \equiv \beta, \quad [\![\bigcirc\tau]\!] \equiv \langle [\![\tau]\!] \rangle^{\alpha}, \quad [\![\tau_1 \to \tau_2]\!] \equiv [\![\tau_1]\!] \to [\![\tau_2]\!] \qquad [\![n]\!] \equiv \alpha^{n}, \quad [\![x_i : \tau_i^{n_i}]\!] \equiv x_i : [\![\tau_i]\!]^{[\![n_i]\!]}$$
$$[\![x]\!] \equiv x, \quad [\![\lambda x.e]\!] \equiv \lambda x.[\![e]\!], \quad [\![e_1\ e_2]\!] \equiv [\![e_1]\!]\ [\![e_2]\!] \quad [\![\mathsf{next}\ e]\!] \equiv \langle [\![e]\!] \rangle \quad [\![\mathsf{prev}\ e]\!] \equiv \tilde{}[\![e]\!]$$

$$
\begin{array}{ll}
\text{Terms} & e \in \mathsf{E} ::= x \mid \lambda x.e \mid e\ e \mid \mathsf{box}\ e \mid \mathsf{unbox}_n\ e \\
\text{Types} & \tau \in \mathsf{T} ::= \beta \mid \tau_1 \to \tau_2 \mid \Box\tau \\
\text{Assignments} & \Gamma \in \mathsf{X} \stackrel{fin}{\to} \mathsf{T} \text{ of types} \\
\text{Stacks} & \Psi \in (\mathsf{X} \stackrel{fin}{\to} \mathsf{T})^* \text{ of type assignments}
\end{array}
$$

$$
\frac{}{\Psi;\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \frac{\Psi;\Gamma, x:\tau_1 \vdash e : \tau_2}{\Psi;\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Psi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Psi;\Gamma \vdash e_2 : \tau_1}{\Psi;\Gamma \vdash e_1 e_2 : \tau_2}
$$

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

Fig. 7. Type system for λ^{S4} [10, Section 4.3]

The translation preserves the typing, i.e. ,

Theorem 7. *If* $\Gamma \overset{n}{\vdash} e : \tau$ *is derivable in* $\lambda^{\bigcirc}$*, then* $[\![\Gamma]\!] \overset{[\![n]\!]}{\vdash} [\![e]\!] : [\![\tau]\!]$ *is derivable in* λ^i*.*

It is easy to prove that the translation preserves the big-step operational semantics.

4.3 Embedding of λ^{S4}

Figure 7 recalls the type system of λ^{S4} [10, Section 4.3]. This calculus is *equivalent* to $\lambda^{\Box}$ [9], but makes explicit use of levels in typing judgments. The operational semantics of λ^{S4} is given indirectly [10, Section 4.3] via the translation into $\lambda^{\Box}$. The embedding of this calculus into λ^i is as follows: the embedding maps types to types:

$$
[\![\Box\tau]\!] \equiv \langle [\![\tau]\!] \rangle \quad [\![\tau_1 \to \tau_2]\!] \equiv [\![\tau_1]\!] \to [\![\tau_2]\!] \quad [\![\beta]\!] \equiv \beta
$$

The embedding on terms is parameterized by a level m:

$$
[\![\mathsf{box}\ e]\!]_m \equiv \mathsf{close}\ \langle [\![e]\!]_{m+1} \rangle \quad [\![\mathsf{unbox}_0\ e]\!]_m \equiv \mathsf{run}\ [\![e]\!]_m \quad [\![\mathsf{unbox}_{n+1}\ e]\!]_{m+n+1} \equiv \%^n(\tilde{}(\mathsf{open}\ [\![e]\!]_m))
$$
$$
[\![x]\!]_m \equiv x \quad [\![\lambda x.e]\!]_m \equiv \lambda x.[\![e]\!]_m \quad [\![e_1 e_2]\!]_m \equiv [\![e_1]\!]_m [\![e_2]\!]_m
$$

The translation of unbox_m depends on the subscript m. unbox_0 corresponds to running code. When $m > 0$ the term unbox_m corresponds to $\tilde{}-$, but if $m > 1$ it also digs into the environment stack to get code from previous stages, and thus the need for the sequence of %s. To define the translation of typing judgments, we must fix a sequence of distinct classifiers $\alpha_1, \alpha_2, \ldots$, and we write A_i for the prefix of the first i classifiers, i.e. , $A_i = \alpha_1, \ldots, \alpha_i$:

$$
[\![\Gamma_0;\ldots;\Gamma_n \vdash e : \tau]\!] \equiv [\![\Gamma_0]\!]^{A_0}, \ldots, [\![\Gamma_n]\!]^{A_n} \overset{A_n}{\vdash} [\![e]\!]_n : [\![\tau]\!]
$$

where $[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]^A \equiv x_1 : [\![\tau_1]\!]^A, \ldots, x_n : [\![\tau_n]\!]^A$. The translation preserves the typing, i.e.

Theorem 8. *If* $\Gamma_0;\ldots;\Gamma_n \vdash e : \tau$ *is derivable in* λ^{S4}*, then* $[\![\Gamma_0;\ldots;\Gamma_n \vdash e : \tau]\!]$ *is derivable in* λ^i*.*

4.4 Relation to Haskell's runST

The typing rules for close and run can be combined into one rule analogous to that for the Haskell's runST [20, 21], namely,

$$
\text{runClosed}\ \frac{\Delta \overset{A}{\vdash} e : \langle\tau\rangle^{\alpha}}{\Delta \overset{A}{\vdash} \mathsf{runClosed}\ e : \tau}\ \alpha \notin \mathrm{FV}(\Delta, A, \tau)
$$

With this rule in place, there is no need to retain the type $\langle\tau\rangle$ and the term $\mathsf{open}\ e$, thus resulting in a proper fragment of λ^i. There is a loss in expressivity, because the embedding of λ^{S4} does not factor through this fragment. In fact, the term $\lambda x.\mathsf{runClosed}\ x$ is not typable, while $\mathsf{run}\ x$ is typable in λ^i_{let} (but $\lambda x.\mathsf{close}\ x$ is still not typable).

5 Implementation

The main motivation for this work was to provide a type system supporting a seamless extension of existing type inference algorithms. In order to assess the practical utility of the approach extended the implementation of MetaOCaml [22] with this new type system. As expected, the extension was a straightforward exercise requiring only minor modifications.[6] More importantly, the presence of an implementation allowed us to perform an initial test of usability of the system from the programmer's point of view. In the example presented in the introduction, no change is needed, because in the implementation, the `.!` construct is taken as the concrete syntax used for the runClosed . For a test suite used in previous work [5], there was also no change required. Although these are only initial tests, they provide indications that the new type system allows the programmer to write safe code in essentially the same way as before, when the `.!` construct was not statically guaranteed to be type safe.

6 Conclusions and future work

We have presented a sound, expressive, and practical type system for functional multi-stage languages. Soundness is demonstrated by establishing type safety. Expressivity is demonstrated by two typability-preserving embeddings of two paradigmatic calculi ($\lambda^{\bigcirc}$ and $\lambda^{\Box}$), as well as by implementing the type system and finding that the type system does not prevent us from expressing existing example MetaOCaml programs. Practicality is demonstrated by showing that it is straightforward to extend the well-established inference algorithms for principal types and principal typings to support classifiers. Furthermore, experience with the implementation and the existing example programs suggests that programmers may not need to significantly change the way they write multi-stage programs.

While our development used a call-by-value (CBV) language, the techniques are equally applicable for a lazy language. Thus, our results can be used, for example, for a multi-stage extension of Haskell.

An important direction for future work will be extending the type system proposed here to the imperative setting. We expect that this will involve incorporating some ideas from previous work on adapting multi-stage languages to the imperative setting [4].

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
2. M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.
3. Alan Bawden. Quasiquotation in LISP. In O. Danvy, editor, *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, San Antonio, 1999. University of Aarhus, Dept. of Computer Science. Invited talk.

[6] In an implementation, there is no need for the two-sort distinction, since the inference algorithm never tries to unify type variables and classifier variables.

4. Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2003. To appear.
5. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
6. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
7. Luís Damas and Robin Milner. Principal type schemes for functional languages. In *9th ACM Symposium on Principles of Programming Languages*. ACM, August 1982.
8. 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.
9. 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.
10. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.
11. Dawson R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 160–170, New York, 1996. ACM Press.
12. 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.
13. 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.
14. Carsten K. Gomard and Neil D. Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
15. 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.
16. J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.
17. Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.
18. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
19. Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components II: Binary-level components. In *[32]*, pages 28–50, 2000.
20. John Launchbury and Simon L. Peyton Jones. State in haskell. *LISP and Symbolic Computation*, 8(4):293–342, 1995. pldi94.
21. John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *Proceedings of the International Conference on Functional Programming*, Amsterdam, 1997.
22. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.cs.rice.edu/~taha/MetaOCaml/, 2001.
23. The MetaML Home Page, 2000. Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.
24. 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.
25. A. Nanevski and F. Pfenning. Meta-programming with names and necessity. submitted, 2003.
26. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.
27. 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.

28. Tim Sheard and Simon Peyton-Jones. Template meta-programming for Haskell. In *Proc. of the Workshop on Haskell*, pages 1–16. ACM, 2002.
29. 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.
30. Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for run-time code generation. *Journal of Functional Programming*, 2003. In [34].
31. Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [27].
32. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
33. 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.
34. Walid Taha, editor. *Journal of Functional Programming, Special Issue on 'Semantics, Applications, and Implementation of Programming Generation (SAIG)'*, volume 13. Cambridge University Press, May 2003.
35. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinksi, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.
36. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
37. 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.
38. Joe Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer-Verlag, 2002.

A Proofs for Type Safety and Embeddings

Proof (Proposition 1). All admissible rules are proved by induction on the (structure of the) derivation of the right-most typing premise. We consider only the most interesting cases:

$\alpha\tau$-subst case close: We know $\Delta \overset{A}{\vdash} \mathsf{close}\, e : \langle\tau\rangle$ and $\alpha \notin \mathrm{FV}(\Delta, A, \tau)$. We must prove $(\Delta \overset{A}{\vdash} \mathsf{close}\ e : \langle\tau\rangle)[\rho]$. Let $\alpha' \notin \mathrm{FV}((\Delta, A, \tau)[\rho])$ and $\rho' = \rho\{\alpha : \alpha'\}$, then by IH applied to the premise of (close), we get $(\Delta \overset{A}{\vdash} e : \langle\tau\rangle^{\alpha})[\rho']$, or, equivalently, $\Delta[\rho] \overset{A[\rho]}{\vdash} e : \langle\tau[\rho]\rangle^{\alpha'}$. Because of the way we have chosen α', we can apply (close) and derive what we want.

$\alpha\tau$-subst case let: Similar to the case (close). Here we have to rename $\overline{\kappa}$ with some $\overline{\kappa}'$ s.t. $\overline{\kappa}' \# \mathrm{FV}((\Delta, A)[\rho])$, and use the fact that $\forall\overline{\kappa}.\tau \equiv \forall\overline{\kappa}'.(\tau[\overline{\kappa} : \overline{\kappa}'])$.

Δ-sub: Straightforward, because $\Delta_1 \succ \Delta_2$ implies $\mathrm{FV}(\Delta_1) \subseteq \mathrm{FV}(\Delta_2)$. Note also that the derivations of premise and conclusion are *structurally* equal, i.e. , the same typing rules are applied.

weaken case close: We must use $\alpha\tau$-subst to rename α (as done in $\alpha\tau$-subst case close) and ensure that (after renaming) $\alpha \# \mathrm{FV}(\sigma_1, A_1)$. Since $\alpha\tau$-subst does not change the structure of the derivation, we can still apply the IH.

weaken case let: We must use $\alpha\tau$-subst to rename $\overline{\kappa}$ (as done in $\alpha\tau$-subst case close) and ensure that (after renaming) $\overline{\kappa} \# \mathrm{FV}(\sigma_1, A_1)$.

e-subst case lam: Since (lam) extends Δ, we must use weaken to extend Δ also in the first premise of (e-subst).

e-subst case let: Similar to the case (lam).

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

$$x \overset{0}{\hookrightarrow} \mathsf{err} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \neq \lambda x.e}{e_1 e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \tilde{}e \overset{0}{\hookrightarrow} \mathsf{err} \qquad \frac{e \overset{0}{\hookrightarrow} v \neq \langle v' \rangle}{\tilde{}e \overset{1}{\hookrightarrow} \mathsf{err}} \qquad \%e \overset{0}{\hookrightarrow} \mathsf{err}$$

$$\frac{e \overset{0}{\hookrightarrow} v \neq \mathsf{close}\ \langle v' \rangle}{\mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v \rangle \quad v \downarrow_0^{\emptyset}\ \text{undefined}}{\mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{0}{\hookrightarrow} v \neq \mathsf{close}\ v'}{\mathsf{open}\ e \overset{0}{\hookrightarrow} \mathsf{err}}$$

Error-propagating rules:

$$\frac{e \overset{n+}{\hookrightarrow} \mathsf{err}}{\lambda x.e \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n}{\hookrightarrow} \mathsf{err}}{e_1 e_2 \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} \mathsf{err}}{e_1 e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v \quad e[x:v] \overset{0}{\hookrightarrow} \mathsf{err}}{e_1 e_2 \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} \mathsf{err}}{e_1 e_2 \overset{n+}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{n+}{\hookrightarrow} \mathsf{err}}{\langle e \rangle \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{n}{\hookrightarrow} \mathsf{err}}{\tilde{}e \overset{n+1}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{n}{\hookrightarrow} \mathsf{err}}{\%e \overset{n+1}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{n}{\hookrightarrow} \mathsf{err}}{\mathsf{run}\ e \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v \rangle \quad v \downarrow_0^{\emptyset} \overset{0}{\hookrightarrow} \mathsf{err}}{\mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{e \overset{n}{\hookrightarrow} \mathsf{err}}{\mathsf{close}\ e \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{n}{\hookrightarrow} \mathsf{err}}{\mathsf{open}\ e \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n}{\hookrightarrow} \mathsf{err}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \quad e_2[x:v] \overset{0}{\hookrightarrow} \mathsf{err}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} \mathsf{err}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \overset{n+}{\hookrightarrow} \mathsf{err}}$$

Fig. 8. Big-step Operational Semantics Extended with Error Generation and Propagation

Figure 8 presents the rules that define the extension of the big-step semantics (Figure 3) with error handling. This extension is part of the definition of type safety.

Proof. (Lemma 1) Promotion is proved by induction on $|A|$, and Demotion is proved by induction on $v \in \mathsf{V}^{|A|+}$. The most interesting case is where Promotion is needed to prove Demotion:

demotion case $\%v \in \mathsf{V}^1$**.** We know $A = \emptyset$ and $\Delta_1, \Delta_2^{\alpha} \overset{\emptyset}{\vdash} v : \tau$.

We must prove $\Delta_1, \Delta_2 \overset{\emptyset}{\vdash} v[\%|\Delta_2|] : \tau$. Let $(\forall \overline{\kappa}_i.\tau_i)^{A_i} = \Delta_2(x_i)$, then by promotion we get $x_i : (\forall \overline{\kappa}_i.\tau_i)^{A_i} \overset{\alpha, A_i}{\vdash} \%_{|A_i|} x_i\ \tau_i$, and we can assume that $\overline{\kappa}_i \# \mathrm{FV}(\Delta_1, \Delta_2, \alpha)$. Thus, we repeatedly apply e-subst to replace x_i with $\%_{|A_i|} x_i$ in $\Delta_1, \Delta_2^{\alpha} \overset{\emptyset}{\vdash} v : \tau$ and derive what we want.

Proof (Theorem 8). By induction over the height of the S4 derivation. An interesting case is the translation of the box term, where the distinctness assumption about the α's is essential for establishing the validity of the side condition for typing the close in the translated term. In the case of unbox_{n+1}, weakening is used, which means that more terms are typable in the target language than in the source language.

B Auxiliary Definitions and Proofs for the Principal Type algorithm

B.1 Most general unifier for many-sorted algebras

Given a many-sorted signature (S, O), i.e. ,

- S is the set of sorts, and
- $O_{\overline{s},s}$ is the set of operations of arity $\overline{s} \to s$, with $\overline{s} \in S^*$

and an S-indexed family $X : S \to \mathbf{Set}$ of *infinite* sets (X_s is the set of variables of sort s), we write:

– $T_s(X)$ for the set of terms of sort s with free variables in X,
– $Sub(X)$ for the set of sort-preserving substitutions $\rho : \prod_{s\in S} X_s \to T_s(X)$
– $\mathrm{FV}(t) \subseteq_{fin} \sum_{s\in S} X_s$ for the set of free variables in $t \in T_s(X)$
– $t[\rho] \in T_s(X)$ for the term obtained by applying the substitution ρ to the term $t \in T_s(X)$.

Given a set $T \subseteq_{fin} \sum_{s\in S} T_s(X) \times T_s(X)$ of equations, we say that $\rho \in Sub(X)$ is a *T-unifier* $\overset{\Delta}{\Longleftrightarrow} t[\rho] = t'[\rho]$ for every $(t, t') \in T$. There exists a function $mgu : \mathcal{P}_{fin}(\sum_{s\in S} T_s(X) \times T_s(X)) \to Sub(X) + \mathsf{fail}$ computing a most general unifier for T, i.e. , mgu has the following properties:

1. If $mgu(T) = \rho$, then ρ is a T-unifier. Moreover,
   (a) $\rho(x) = x$ for $x \notin \mathrm{FV}(T)$
   (b) $\mathrm{FV}(T[\rho]) \subseteq \mathrm{FV}(T)$
   (c) the operations occurring in ρ occur in T already
2. If ρ' is a T-unifier, then $\rho = mgu(T)$ is defined and there exists $\rho'' \in Sub(X)$ s.t. $\rho'_s(x) = \rho_s(x)[\rho'']$ for any $s \in S$ and $x \in X_s$.

Remark 3. The type inference algorithms for λ^i and λ^i_{let} use mgu for a signature with two sorts, A for classifiers and T for types, and the following operations:

$$fun : \mathsf{T}, \mathsf{T} \to \mathsf{T} \qquad code : \mathsf{T}, \mathsf{A} \to \mathsf{T} \qquad ccode : \mathsf{T} \to \mathsf{T}$$

We write α for classifier variables, β for type variables, and τ for terms of sort T of types. The only terms of sort A are variables α. Moreover, we write $\tau_1 \to \tau_2$ for $fun(\tau_1, \tau_2)$, $\langle\tau\rangle^\alpha$ for $code(\tau, \alpha)$ and $\langle\tau\rangle$ for $ccode(\tau)$.

Proof (Theorem 5). The proof is by induction on the computation of $W(\Delta, A, e, \mathcal{K})$. We consider case let.

let We have $W(\Delta, A, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \mathcal{K}) = (\rho_2\rho_1, \tau_2, \mathcal{K}'')$ and $(\rho_1, \tau_1, \mathcal{K}') = W(\Delta, A, e_1, \mathcal{K})$ and $(\rho_2, \tau_2, \mathcal{K}'') = W(\Delta[\rho_1]\{x : \mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1])^{A[\rho_1]}\}, A[\rho_1], e_2, \mathcal{K}')$.
Induction hypothesis on e_1 and e_2 gives:
– $\mathcal{K} \subseteq \mathcal{K}' \subseteq \mathcal{K}''$
– $\Delta[\rho_1] \overset{A[\rho_1]}{\vdash} e_1 : \tau_1$
– $\mathrm{FV}(\tau_1, \Delta[\rho_1], A[\rho_1]) \subseteq \mathcal{K}'$
– $\Delta'[\rho_2] \overset{A[\rho_2\rho_1]}{\vdash} e_2 : \tau_2$ with $\Delta' \equiv \Delta[\rho_1]\{x : \mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1])^{A[\rho_1]}\}$
– $\mathrm{FV}(\tau_2, \Delta'[\rho_2], A[\rho_2\rho_1]) \subseteq \mathcal{K}''$
We show the following:
– $\Delta[\rho_2\rho_1] \overset{A}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2$
  Let ρ' be a renaming of $\mathrm{FV}(\tau_1) - \mathrm{FV}(\Delta[\rho_1], A[\rho_1])$ with fresh variables so that $\Delta[\rho'\rho_1] \equiv \Delta[\rho_1]$ and $A[\rho'\rho_1] = A[\rho_1]$ and $\mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1])[\rho_2] \equiv \mathrm{close}(\tau_1[\rho_2\rho'], \Delta[\rho_2\rho_1], A[\rho_2\rho_1])$.
  Then we have $\Delta[\rho_2\rho_1]\{x : (\mathrm{close}(\tau_1[\rho_2\rho'], \Delta[\rho_2\rho_1], A[\rho_2\rho_1]))^{A[\rho_2\rho_1]}\} \overset{A[\rho_2\rho_1]}{\vdash} e_2 : \tau_2$. Proposition 1 gives $\Delta[\rho_2\rho'\rho_1] \overset{A[\rho_2\rho'\rho_1]}{\vdash} e_1 : \tau_1[\rho_2\rho']$, hence $\Delta[\rho_2\rho_1] \overset{A[\rho_2\rho_1]}{\vdash} e_1 : \tau_1[\rho_2\rho']$. Finally, $\Delta[\rho_2\rho_1] \overset{A[\rho_2\rho_1]}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2$ by (let).
– $\mathcal{K} \subseteq \mathcal{K}''$
– $\mathrm{FV}(\tau_2, \Delta[\rho_2\rho_1], A[\rho_2\rho_1]) \subseteq \mathrm{FV}(\tau_2, \Delta'[\rho_2], A[\rho_2\rho_1]) \subseteq \mathcal{K}''$

The following lemma is used for the proof of completeness (Theorem 6) case let.

Lemma 2. $\mathrm{close}(\tau, \Delta, A)[\rho] \succ \forall\overline{\kappa}.(\tau[\rho])$ *provided* $\mathrm{FV}(\Delta[\rho], A[\rho])\#\overline{\kappa}$.

Proof. Let $\overline{\kappa}_1 = \mathrm{FV}(\Delta, A)$, then $\mathrm{FV}(\overline{\kappa}_1[\rho])\#\overline{\kappa}$. Let $\overline{\kappa}_2 = \mathrm{FV}(\tau) - \{\overline{\kappa}_1\}$, and let $\overline{\kappa}'_2$ be fresh variables for renaming the variables $\overline{\kappa}_2$. Then $\mathrm{close}(\tau, \Delta, A)[\rho] \equiv (\forall\overline{\kappa}_2.\tau)[\rho] \equiv (\forall\overline{\kappa}'_2.\tau')[\rho] \equiv \forall\overline{\kappa}'_2.(\tau'[\rho])$ with $\tau' = \tau[\overline{\kappa}_2 : \overline{\kappa}'_2]$. We have $\mathrm{FV}(\forall\overline{\kappa}'_2.(\tau'[\rho])) \subseteq \mathrm{FV}(\overline{\kappa}_1[\rho])\#\overline{\kappa}$.

Since $\tau'[\rho][\overline{\kappa}'_2 : \overline{\kappa}_2] = \tau[\rho]$ and $[\overline{\kappa}'_2 : \overline{\kappa}_2]$ has support $\overline{\kappa}'_2$, by definition of $\succ$ we can conclude $\forall\overline{\kappa}'_2.(\tau'[\rho]) \succ \forall\overline{\kappa}.(\tau[\rho])$.

Proof (Theorem 6). The proof is by induction on the structure of e. We consider two cases let and $(\alpha)e$.

$$\mathsf{let}\ \frac{\Delta[\rho'] \overset{A[\rho']}{\vdash} e_1 : \tau'_1 \quad \Delta[\rho']\{x : (\forall\overline{\kappa}.\tau'_1)^{A[\rho']}\} \overset{A[\rho']}{\vdash} e_2 : \tau'_2}{\Delta[\rho'] \overset{A[\rho']}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau'_2}\ \mathrm{FV}(\Delta[\rho'], A[\rho'])\#\overline{\kappa}$$

By IH $(\rho_1, \tau_1, \mathcal{K}') = W(\Delta, A, e_1, \mathcal{K})$ and exists $\rho'' \in \mathsf{Sub}$ s.t. $\tau'_1 = \tau_1[\rho'']$ and $\Delta[\rho'] \equiv \Delta[\rho''\rho_1]$ and $A[\rho'] = A[\rho''\rho_1]$.

- $\Delta[\rho''\rho_1]\{x : (\forall\overline{\kappa}.\tau_1[\rho''])^{A[\rho''\rho_1]}\} \overset{A[\rho''\rho_1]}{\vdash} e_2 : \tau'_2$
- $(\mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1]))[\rho''] \succ \forall\overline{\kappa}.\tau_1[\rho'']$ by Lemma 2 since $\mathrm{FV}(\Delta[\rho''\rho_1], A[\rho''\rho_1])\#\overline{\kappa}$
- $\Delta[\rho''\rho_1]\{x : (\mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1]))[\rho'']^{A[\rho''\rho_1]}\} \overset{A[\rho''\rho_1]}{\vdash} e_2 : \tau'_2$ by Proposition 1

By IH $(\rho_2, \tau_2, \mathcal{K}'') = W(\Delta', A, e_2, \mathcal{K}')$ and exists $\rho''' \in \mathsf{Sub}$ s.t. $\tau'_2 = \tau_2[\rho''']$ and $\Delta'[\rho''] \equiv \Delta'[\rho'''\rho_2]$ and $A[\rho''\rho_1] = A[\rho'''\rho_2\rho_1]$ with $\Delta' \equiv \Delta[\rho_1]\{x : \mathrm{close}(\tau_1, \Delta[\rho_1], A[\rho_1])^{A[\rho_1]}\}$.

To conclude we show:

- $W(\Delta, A, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \mathcal{K}) = (\rho_2\rho_1, \tau_2, \mathcal{K}'')$
- $\tau'_2 = \tau_2[\rho''']$
- $\Delta[\rho'] \equiv \Delta[\rho''\rho_1] \equiv \Delta[\rho'''\rho_2\rho_1]$ and $A[\rho'] = A[\rho''\rho_1] = A[\rho'''\rho_2\rho_1]$

$$\mathsf{close}\ e\ \frac{\Delta[\rho'] \overset{A[\rho']}{\vdash} e : \langle\tau'_1\rangle^{\alpha}}{\Delta[\rho'] \overset{A[\rho']}{\vdash} \mathsf{close}\ e : \langle\tau'_1\rangle}\ \alpha \notin \mathrm{FV}(\Delta[\rho'], A[\rho'], \tau'_1)$$

By IH $W(\Delta, A, e, \mathcal{K}) = (\rho, \tau, \mathcal{K}')$ and exists $\rho'' \in \mathsf{Sub}$ s.t. $\langle\tau'_1\rangle^{\alpha} = \tau[\rho'']$, and $\Delta[\rho'] \equiv \Delta[\rho''\rho]$ and $A[\rho'] = A[\rho''\rho]$.

- $\mathrm{FV}(\tau, \Delta[\rho], A[\rho]) \subseteq \mathcal{K}'$ by Theorem 5

Take $\alpha', \beta \notin \mathcal{K}'$, so that $\alpha', \beta\#\mathrm{FV}(\tau, \Delta[\rho], A[\rho])$, and let $\rho''' = \rho''\{\beta : \tau'_1, \alpha' : \alpha\}$, then

- $\tau[\rho'''] = \langle\beta\rangle^{\alpha'}[\rho'''] = \langle\tau'_1\rangle^{\alpha}$
- $\rho_1 = mgu(\tau, \langle\beta\rangle^{\alpha'})$ is defined, and $\rho''' = \rho_2\rho_1$ for some ρ_2.
- $\alpha'[\rho_1] \notin \mathrm{FV}(\Delta[\rho_1\rho], A[\rho_1\rho], \beta[\rho_1])$ since $\alpha = \alpha'[\rho_2\rho_1] \notin \mathrm{FV}(\Delta[\rho_2\rho_1\rho], A[\rho_2\rho_1\rho], \beta[\rho_2\rho_1]) = \mathrm{FV}(\Delta[\rho'''\rho], A[\rho'''\rho], \beta[\rho'''\rho]) = \mathrm{FV}(\Delta[\rho'], A[\rho'], \tau'_1)$

To conclude, we show:

- $W(\Delta, A, \mathsf{close}\ e, \mathcal{K}) = (\rho_1\rho, \langle\beta[\rho_1]\rangle, \mathcal{K}' \cup \{\beta, \alpha'\})$ is defined
- $\langle\tau'_1\rangle = \langle\beta[\rho_1]\rangle[\rho_2]$ since $\beta[\rho_2\rho_1] = \beta[\rho'''] = \tau'_1$
- $\Delta[\rho'] \equiv \Delta[\rho_2\rho_1\rho]$ and $A[\rho'] = A[\rho_2\rho_1\rho]$