A Contextual Account of Staged Computations

Matthias Puech

Inria, France*
`matthias.puech@inria.fr`

Abstract

Programming languages that allow us to manipulate code as data pose a notorious challenge to type system designers. We propose contextual type theory with first-class environments as a foundational typing discipline for multi-stage functional programming: it strictly subsumes previous proposals while being based on a Curry-Howard correspondence with modal logic. In particular, we show a conservative embedding of Taha and Nielsen's environment classifiers into it. This embedding sheds a new light on environment classifiers as approximations over environments, and on the relationship between modal logics.

1 Introduction

For the working programmer like for the programming language theorist, there is a deep duality between code and data that arises in many contexts: for instance, many algorithms can be understood as code-generating programs (e.g., parser generators), and higher-order programs can be understood as programs manipulated as data (e.g., defunctionalization).

Multi-staging is a feature of functional programming languages which gives the user the illusion to manipulate code snippets. It provides a unifying abstraction to support partial evaluation[10], macro-expansion [6] and more [15]. The main idea dates back to at least Lisp's quasi-quotations, and is threefold: *(i)* a special *quote* operator (written "`‘`") turns an expression into its syntactical representation—an S-expression; *(ii)* inside a quote, the *unquote* operator (written "`,`") allows us to escape the quotation and refer to another computed expression whose value will be plugged at this place; and *(iii)* an evaluation primitive ("`run`") strips a piece of code from its quotes and evaluates it. For example in Lisp, the function:

```
(lambda (t) ‘(lambda (x) (,t (+ x y))))
```

returns an unevaluated S-expression with its argument `t` plugged in; applied to symbol `‘square`, it returns:

```
‘(lambda (x) (square (+ x y)))
```

which can then be evaluated with the `run` primitive, provided that `y` is bound. Effectively, multi-staging organizes the evaluation of expressions into user-specified *stages*, providing a fine control over evaluation than the bare strategies. Multi-staging refers to the fact that quotes (and thus unquotes) can be nested arbitrarily deeply: we can write code that will evaluate to code that will in turn evaluate to, e.g., a function.

Type systems for multi-staging

One can express optimizations with multi-staging that would be much less direct without it [10]. However, the added expressiveness makes it difficult to write correct programs: code and values can be mixed up, variables can be captured or escape their scope, etc. Early on,

* This work was carried out while at McGill University, Dept. of Computer Science, Canada.

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

type systems have been proposed to statically detect these potential bugs [10]. In particular, Davies and Pfenning's work on type systems for multi-staged languages was the first to be logically grounded, i.e., based on a Curry-Howard correspondence with the modal logic S4 [5] and linear-time temporal logic [4], respectively. In both cases, the modality $\Box A$ or $\bigcirc A$ denotes the type of a code snippet which, once evaluated, would lead to a value of type A; quote and unquote respectively introduce and eliminate the modality.

- The first system, $\lambda\Box$ [5], is a λ-calculus with (anti-)quotations and an evaluation primitive, but had the limitation that all quoted code snippets must be closed, i.e., must not contain free variables like `y` in the example above. This made programming in $\lambda\Box$ rather cumbersome.
- The second system, $\lambda\bigcirc$ [4], lifted this restriction while maintaining (a staged notion of) lexical scoping, but lost the ability to guarantee safe evaluation of the code: evaluating an expression containing free variables would give a run-time error.

Since then, much work attempted at reconciling these features, i.e., manipulate open code while guaranteeing safe evaluation [16, 15, 2, 7, 13]. Notably, Taha and Nielsen's λ^{α} [15], which, using a seminal notion of "stage delimiters" called *environment classifiers*, provided $\lambda\bigcirc$ with safe evaluation. Unfortunately, the logical correspondence was lost, and the operational semantics made complex by a strong reduction discipline and a context-sensitive grammar of values.

Contextual types for multi-staging

In this paper, we revisit the initial dilemma in the light of recent work on contextual types [9, 11, 1]. These generalize the necessity modality $\Box A$ into a contextual modality $[\Gamma.\, A]$ which denotes the type of expressions of type A whose free variables are taken from environment Γ. We propose contextual type theory, augmented with first-class environments, as a foundation for typing multi-stage functional programs; we interpret contextual types as the types of code snippets. More specifically, we describe λ^{ctx}, the contextual λ-calculus with first-class environments, that allows the program to manipulate open code and nonetheless guarantees safe evaluation. It has a simple operational semantics with a context-free grammar of values, thanks to a notion of meta-substitution, and its type system is in direct Curry-Howard correspondence with contextual logic. λ^{ctx} is a core calculus but following Davies & Pfenning we give an "implicit" variant, λ_I^{ctx}, which follows more closely the programming practices.

▶ **Example 1.** In λ_I^{ctx}, the Lisp code above would be written (in a pseudo-ML syntax):

```
let f t = [y. fun x → ~t (x+y)]
```

Brackets `[]` denote quoting, and `~` denotes unquoting. The open code `fun x → ~t (x+y)` comes together with the environment in which it is meaningful, `y`; `y` is bound in the code. Now, the fact that the argument `t` must be code representing a function, and that the result has one more free variable of type `int` than the argument are all encoded in the type:

```
val f : [γ. int → α] → [γ, int. int → α]
```

Brackets denote code type, and γ is an *environment variable* which stands for any concrete environment; it is implicitly bound at the top level, like type variable α. Given such an `int`, say, 2, we can substitute (primitive `subst`) it for `y` and evaluate (primitive `run`) the ground result safely on, say, 3. This expression evaluates to 25:

```
run (subst (f [. fun x → x * x]) [. 2]) 3
```

Embedding environment classifiers

Our calculus, while based on $\lambda\Box$, subsumes and refines λ^α: we give a new embedding of λ^α into λ_I^{ctx}, therefore providing a logical basis to environment classifiers; it is our main technical contribution. At its core, this transformation infers missing environment information from a λ^α typing derivation to obtain a family of implicit derivations in λ_I^{ctx}. Notably, the translation is presented declaratively, needs to translate whole typing derivations at once, and introduces constrained *logic* variables to represent families of target derivations; showing the translation correct therefore amounts to prove that it always returns a derivation family (with constrained logic variables) that can be instantiated into a proper derivation, respecting the constraints. Beyond mere expressiveness, this embedding also suggests a novel two-zone presentation of temporal logic, thanks to the combination of contextual types and quantification over environments.

Contributions

Our contributions are the following:

- We define the syntax, typing and semantics of λ^{ctx}, a core calculus able to express safe staged computations (Sec. 2). We present its implicit counterpart, λ_I^{ctx}, which is closer to programming practices.
- We present a type-preserving embedding from λ^α into λ_I^{ctx} that infers missing environment annotations and generates constraints. We sketch the proof that it preserves types up to annotations, and that it is decidable, i.e., that constraints always have a solution (Sec. 3.2). This is our main technical contribution.

2 Contextual types with first-class environments

One possible type system design starts with the necessity modality $\Box A$ of the modal logic S4, that is interpreted as the type of code snippets of type A [5]. The resulting type system, $\lambda\Box$, has two distinct environments, Δ and Γ, corresponding respectively to necessary and true hypotheses, and two associated sets of variables, *meta-variables* u and *usual variables* x (this presentation is called *two-zone*). The *box* construct $[M]$ lets us build a piece of code; it introduces the $\Box$ modality. Inside M, the meta-variables of Δ can be used, but no free ordinary variable is allowed; this restriction corresponds to the principle that a necessary proposition should only depend on necessary hypotheses. The *let-box* construct $\mathsf{let}_\Box\, u = M \,\mathsf{in}\, N$ lets us name a piece of code M as meta-variable u inside N; it is the elimination form of the $\Box$ type. The resulting λ-calculus accepts a safe evaluation primitive $\mathtt{run} : \Box A \to A$, which corresponds to the reflexivity principle of S4. This language is expressive enough to encode examples such as a staged regular expression matcher [5].

However, code of type $\Box A$ is necessarily closed (no free ordinary variable allowed), which makes programming in this language cumbersome. Contextual types [9] generalize this $\Box$ modality by allowing free variables in code, but explicitly keeping track of them in types: a value of type $[\Gamma.\, A]$ is a piece of code with free variables taken only from environment Γ. We now write $[\Gamma.\, M]$ for the introduction form, and since a piece of code might now be open, we have to rebind its free variables each time we refer to one: meta-variable instances have to carry a substitution $u\{\sigma\}$ replacing each free variable of u by a term of the right type. For instance, the function $\lambda f.\, \mathsf{let}_\Box\, u = f \,\mathsf{in}\, [x, y.\, (u\{x\})\, y]$ wraps a piece of code with one free variable in an application node with a new free variable y (the returned code has two free

variables); this function has type $[p.\, q \to r] \to [p, q.\, r]$. The resulting programming language was proposed to express staged programs more naturally [9].

Nevertheless, explicitly mentioning all free variables can hinder modularity. For instance, the previous function takes code with exactly one free variable of type p, whereas any open code with at least one variable of type p could be applied to it: the function could be parametric in its environment. First-class environments [11] allow us to quantify over and instantiate prefixes of environments: a value of type $\forall\alpha.\, A$, introduced by the syntactic construct $\Lambda\alpha.\, M$, is a value of type A where the *environment variable* α can be used in place of any environment or environment prefix; such a variable can be replaced by any concrete environment, thanks to the elimination form: $M\,\Gamma$. Now, a substitution σ can be either concrete, or map all remaining variables to themselves, i.e., be the identity id_α. For instance, function $\Lambda\alpha.\, \lambda f.\, \mathsf{let}_\Box\, u = f\ \mathsf{in}\ [\alpha, y.\, u\{\mathsf{id}_\alpha, x\}\, y]$ now has type $\forall\alpha.\, [\alpha.\, q \to r] \to [\alpha, q.\, r]$.

2.1 The contextual λ-calculus λ^{ctx}

Let us formally define this language, that we call λ^{ctx}. Its syntax is defined by the grammar at the top of Fig. 1. We introduce four disjoint sets of variables: base types p, q, r, environment variables α, β, γ, usual variables x, y, z, f, g, h and meta-variables u, v, w. We prefer to write minimum information in environments and substitutions, and to rely on positions to match them: we write $\tilde{\Gamma}$ for an environment stripped of the variable names, and $\hat{\Gamma}$ for an environment stripped of the types. For instance, if $\Gamma = (\alpha, x : A, y : B)$ then $\tilde{\Gamma} = \alpha, A, B$ and $\hat{\Gamma} = \alpha, x, y$. For the same reason, substitutions do not carry variable names; they are matched positionally. Naturally, $\lambda x.\, M$ binds variable x in M, $\mathsf{let}_\Box\, u = M\ \mathsf{in}\ N$ binds meta-variable u in N, $\Lambda\alpha.\, M$ binds environment variable α in M; also $[\hat{\Gamma}.\, M]$ binds all variables of $\hat{\Gamma}$ in M. Note that contexts (resp. substitutions) start with an environment variable (resp. identity substitution), called its *bottom variable*, which can be instantiated to a concrete environment. An *environment context* G is an environment with a *hole* $[]$, standing for another environment. We write $G(\Psi)$ for the operation of plugging environment Ψ into the hole of G.

Its type system can be found on Fig. 1. The meta-environment Δ in the typing judgment contains only meta-variables, whose types start with a box $[\tilde{\Gamma}.\, A]$. The environment Γ contains all usual variables. It has to have a bottom variable; we distinguish a special "top-level" environment variable α_0. The first three rules Var, Lam and App are standard. The Box rule replaces the current environment Γ by Ψ, discarding all previously available (usual) variables. LetBox promotes a value of box type into a meta-variable. When referring to a meta-variable $u :: [\tilde{\Psi}.\, A]$, one has to provide a substitution σ mediating between Ψ and the current environment Γ (rule Meta). When generalizing over α, the Gen rule ensures that α was not used, to avoid a variable clash. In the remainder, $\mathrm{FV}(X)$ denotes the set of free variables of X (in the usual sense). The Inst rule uses the substitution of environment variable to instantiate an environment variable to the given concrete environment:

▶ **Definition 2** (Environment substitution)**.** The substitution of an environment in a type $\{\Psi/\alpha\}A$, term $\{\Psi/\alpha\}M$, substitution $\{\Psi/\alpha\}\sigma$ and environment $\{\Psi/\alpha\}\Gamma$ is defined as follows (omitted cases are homomorphic):

$$\{\Psi/\alpha\}(\forall\beta.\, A) = \forall\beta.\, \{\Psi/\alpha\}A \qquad \beta \notin \mathrm{FV}(\Psi) \tag{1}$$

$$\{\Psi/\alpha\}[\tilde{\Gamma}.\, A] = [\{\Psi/\alpha\}\tilde{\Gamma}.\, \{\Psi/\alpha\}A] \tag{2}$$

$$\{\Psi/\alpha\}(\Gamma, x : A) = \{\Psi/\alpha\}\Gamma, x : \{\Psi/\alpha\}A \qquad x \notin \mathrm{FV}(\Psi) \tag{3}$$

$$\{\Psi/\alpha\}\beta = \beta \qquad \beta \neq \alpha \tag{4}$$

$$
\begin{aligned}
A, B &::= p \mid A \to B \mid [\tilde{\Gamma}.\, A] \mid \forall \alpha.\, A && \text{Type}\\
\Gamma, \Psi &::= \alpha \mid \Gamma, x : A && \text{Environment}\\
M, N &::= x \mid \lambda x.\, M \mid M\, N \mid [\hat{\Gamma}.\, M] \mid \mathsf{let}_{\square}\, u = M\, \mathsf{in}\, N \mid u\{\sigma\} \mid \Lambda \alpha.\, M \mid M\, \Gamma && \text{Term}\\
\sigma &::= \mathsf{id}_\alpha \mid \sigma, M && \text{Substitution}\\
\Delta, \Theta &::= \cdot \mid \Delta, u :: [\tilde{\Gamma}.\, A] && \text{Meta-environment}\\
G &::= [] \mid G, x : A && \text{Environment context}
\end{aligned}
$$

$\boxed{\Delta; \Gamma \vdash M : A}$ Term M has type A in environment Γ and meta-environment Δ

$$
\textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A} \qquad
\textsc{Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x.\, M : A \to B} \qquad
\textsc{App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\, N : A \to B}
$$

$$
\textsc{Box}\ \frac{\Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash [\hat{\Psi}.\, M] : [\Psi.\, A]} \qquad
\textsc{LetBox}\ \frac{\Delta; \Gamma \vdash M : [\tilde{\Psi}.\, A] \qquad \Delta, u :: [\tilde{\Psi}.\, A]; \Gamma \vdash N : B}{\Delta; \Gamma \vdash \mathsf{let}_{\square}\, u = M\, \mathsf{in}\, N : B}
$$

$$
\textsc{Meta}\ \frac{u :: [\tilde{\Psi}.\, A] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \tilde{\Psi}}{\Delta; \Gamma \vdash u\{\sigma\} : A} \qquad
\textsc{Gen}\ \frac{\Delta; \Gamma \vdash M : A \qquad \alpha \notin \mathrm{FV}(\Delta, \Gamma)}{\Delta; \Gamma \vdash \Lambda \alpha.\, M : \forall \alpha.\, A}
$$

$$
\textsc{Inst}\ \frac{\Delta; \Gamma \vdash M : \forall \alpha.\, A}{\Delta; \Gamma \vdash M\, \Psi : \{\Psi/\alpha\} A}
$$

$\boxed{\Delta; \Gamma \vdash \sigma : \tilde{\Psi}}$ Substitution σ transports terms from environment Ψ to environment Γ

$$
\textsc{Id}\ \frac{}{\Delta; G(\alpha) \vdash \mathsf{id}_\alpha : \alpha} \qquad
\textsc{Cons}\ \frac{\Delta; \Gamma \vdash \sigma : \tilde{\Psi} \qquad \Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash (\sigma, M) : (\tilde{\Psi}, A)}
$$

Figure 1 Syntax and typing of λ^{ctx}, the contextual λ-calculus

$$
\{\Psi/\alpha\}\alpha = \Psi \tag{5}
$$

$$
\{\Psi/\alpha\}[\hat{\Gamma}.\, M] = [\{\Psi/\alpha\}\hat{\Gamma}.\, \{\Psi/\alpha\}M] \qquad \mathrm{FV}(\Psi) \not\subset \mathrm{FV}(\Gamma) \tag{6}
$$

$$
\{\Psi/\alpha\}(\Lambda \beta.\, M) = \Lambda \beta.\, \{\Psi/\alpha\}M \qquad \beta \neq \alpha \tag{7}
$$

$$
\{\Psi/\alpha\}\mathsf{id}_\beta = \mathsf{id}_\beta \qquad \beta \neq \alpha \tag{8}
$$

$$
\{\Psi/\alpha\}\mathsf{id}_\alpha = \mathsf{id}(\Psi) \tag{9}
$$

It is capture-avoiding, *cf.* the side condition in (1). More importantly, substituting an environment for a variable in a term may change the binding structure of a term; for instance, $\{\beta, x : p, y : q/\alpha\}[\alpha, z.\, M] = [\beta, x, y, z.\, \{\beta, x : p, y : q/\alpha\}M]$. This fact explains the side-conditions in (3) and (6). In (9), when substituting an environment for an identity subsitution, we need to expand this identity substitution. This operation is defined as follows:

$$V ::= \lambda x.\, M \mid [\hat{\Gamma}.\, M] \mid \Lambda\alpha.\, V \qquad \text{Value}$$

$\boxed{M \downarrow V}$ Term M evaluates to value V

$$\textsc{Lam}\ \frac{}{\lambda x.\, M \downarrow \lambda x.\, M} \qquad \textsc{App}\ \frac{M \downarrow \lambda x.\, M' \qquad N \downarrow V \qquad \{V/x\}M' \downarrow V'}{M\, N \downarrow V'} \qquad \textsc{Box}\ \frac{}{[\hat{\Psi}.\, M] \downarrow [\hat{\Psi}.\, M]}$$

$$\textsc{LetBox}\ \frac{M \downarrow [\hat{\Psi}.\, M'] \qquad \{\!\{\hat{\Psi}.M'/u\}\!\}N \downarrow V}{\mathsf{let}_{\Box}\, u = M\, \mathsf{in}\, N \downarrow V} \qquad \textsc{Gen}\ \frac{M \downarrow V}{\Lambda\alpha.\, M \downarrow \Lambda\alpha.\, V} \qquad \textsc{Inst}\ \frac{M \downarrow \Lambda\alpha.\, V}{M\, \Psi \downarrow \{\hat{\Psi}/\alpha\}V}$$

Figure 2 Big-step operational semantics of λ^{ctx}

▶ **Definition 3** (Identity substitution expansion). $\mathsf{id}(\hat{\Gamma})$ is defined recursively on $\hat{\Gamma}$:

$$\mathsf{id}(\alpha) = \mathsf{id}_{\alpha} \qquad \mathsf{id}(\hat{\Gamma}, x) = \mathsf{id}(\hat{\Gamma}), x \tag{10}$$

The Cons typing rule matches each component of a substitution with each component of the environment, positionally. The Id rule features built-in weakening, expressed using environment contexts substitution $G(\alpha)$: the identity substitution id_{α} mediates between the environment that is just the variable α, and any environment $G(\alpha)$ having α as bottom variable; all other declarations in G are weakened.[1]

We illustrate the simplicity of λ^{ctx} by describing its operational semantics on Fig. 2. Values are described by the context-free grammar at the top. They include λ-abstractions, generalizations and boxes. Note that any boxed term is a value; this reflects that we never evaluate a piece of code (unless it is run). Values are a subset of terms; hereafter, we allow to consider implicitly values as terms. The LetBox rule substitutes the definiens of u in the body N, using the substitution of a meta-variable into a term $\{\!\{\hat{\Psi}.N/u\}\!\}M$:

▶ **Definition 4** (Meta-substitution). The substitution of a meta-variable in a term $\{\!\{\hat{\Psi}.N/u\}\!\}M$ and substitution $\{\!\{\hat{\Psi}.N/u\}\!\}\sigma$ is defined as follows (omitted cases are homomorphic):

$$\{\!\{\hat{\Psi}.N/u\}\!\}(v\{\sigma\}) = v\{\{\!\{\hat{\Psi}.N/u\}\!\}\sigma\} \qquad \text{if } u \neq v \tag{11}$$

$$\{\!\{\hat{\Psi}.N/u\}\!\}(u\{\sigma\}) = \{\{\!\{\hat{\Psi}.N/u\}\!\}\sigma\}_{\hat{\Psi}}N \tag{12}$$

When the meta-substitution arrives to its meta-variable $u\{\sigma\}$, it triggers a simultaneous substitution $\{\sigma\}_{\hat{\Psi}}N$, defined as follows:

▶ **Definition 5** (Simultaneous substitution). The simultaneous substitution of σ and $\hat{\Psi}$ in a term $\{\sigma\}_{\hat{\Psi}}M$ and substitution $\{\sigma\}_{\hat{\Psi}}\sigma'$ is defined as follows (omitted cases are homomorphic):

$$\{\sigma\}_{\hat{\Psi}}(\lambda x.\, M) = \lambda x.\, \{\sigma, x\}_{\hat{\Psi},x}M \qquad x \notin \mathrm{FV}(\hat{\Psi}) \cup \mathrm{FV}(\sigma) \tag{13}$$

$$\{\sigma\}_{\hat{\Psi}}[\hat{\Gamma}.\, M] = [\hat{\Gamma}.\, M] \tag{14}$$

$$\{\sigma\}_{\hat{\Psi}}(u\{\sigma'\}) = u\{\{\sigma\}_{\hat{\Psi}}\sigma'\} \tag{15}$$

[1] Instead of $G(\alpha)$, we could have written the more informal $\alpha, x_1 : A_1, \ldots, x_n : A_n$; the notion of environment context will be instrumental below, in Sec. 3.2.

$$\{\sigma, M\}_{\hat{\Psi},y} x = \{\sigma\}_{\hat{\Psi}} x \qquad x \neq y \tag{16}$$

$$\{\sigma, M\}_{\hat{\Psi},x} x = M \tag{17}$$

It is the standard definition, except for the facts that *(i)* because of our representation choice, variable names and substituens are found in respectively $\hat{\Psi}$ and σ and must be matched positionally, as done in the last three equations, and *(ii)* it stops at the boundary of boxes $[\hat{\Gamma}.\, M]$, since these do not contain any free variables except for those in $\hat{\Gamma}$.

The first two evaluation rules are standard for the call-by-value weak head-reduction of the λ-calculus. They rely on single substitution, which is easily defined in terms of simultaneous substitution:

▶ **Definition 6** (Single substitution)**.** The substitution of a single usual variable in a term $\{N/x\}M$ is defined by $\{\mathsf{id}_\alpha, N\}_{\alpha,x} M$.

Running code

Now, λ^{ctx} can be extended with a `run` primitive of type $(\forall\alpha.\,[\alpha.\, A]) \to A$: if a piece of code is well-typed in an environment that is universally quantified, then it cannot have free variables; therefore it can safely be taken for its semantic value, i.e., stripped from its box. In other words, `run` evaluates as follows:

$$\frac{M \downarrow \Lambda\alpha.\,[\alpha.\, N] \qquad N \downarrow V}{\texttt{run}\ M \downarrow V}$$

Since we have control over single free variables, we can also define a substitution primitive, which replaces one free variable in a code value. This principle is definable:

$$\texttt{subst} : \forall\alpha.\,[\alpha, A.\, B] \to [\alpha.\, A] \to [\alpha.\, B] \tag{18}$$

$$= \Lambda\alpha.\,\lambda xy.\,\mathsf{let}_\square\, u = x\ \mathsf{in}$$

$$\mathsf{let}_\square\, v = y\ \mathsf{in}\ u\{\mathsf{id}_\alpha, v\{\mathsf{id}_\alpha\}\} \tag{19}$$

▶ **Example 7.** Consider the following terms:

1. The two-level η-expansion [3] $\lambda f.\,\Lambda\alpha.\,\mathsf{let}_\square\, u = f\ (\alpha, x : p)\,[\alpha, x.\, x]\ \mathsf{in}\ [\alpha.\,\lambda x.\, u\{\mathsf{id}_\alpha, x\}]$ has type $(\forall\alpha.\,[\alpha.\, p] \to [\alpha.\, q]) \to \forall\alpha.\,[\alpha.\, p \to q]$. The first arrow, at the current stage (outside any box), gets turned into the last arrow, at the next stage (inside a box): this function reifies the function space of its argument into the code of a function.
2. A code value representing a function, of type $[\alpha.\, p \to q]$, can be converted to a code value with a free variable, of type $[\alpha, p.\, q]$; the coercion $\Lambda\alpha.\,\lambda x.\,\mathsf{let}_\square\, u = x\ \mathsf{in}\,[\alpha, y.\, u\{\mathsf{id}_\alpha\}\ y]$ applies the free variable to the function. Conversely, we turn open code into a function by substituting its free variable by the function argument: $\Lambda\alpha.\,\lambda x.\,\mathsf{let}_\square\, u = x\ \mathsf{in}\,[\alpha.\,\lambda y.\, u\{\mathsf{id}_\alpha, y\}]$.
3. Example 1 from the introduction is written in λ^{ctx}:

   $$\mathbf{T} = \Lambda\alpha.\,\lambda t.\,\mathsf{let}_\square\, u = t\ \mathsf{in}\,[\alpha, y.\,\lambda x.\, u\{\mathsf{id}_\alpha\}\ (x + y)]$$

   It has type $\forall\alpha.\,[\alpha.\, \mathit{int} \to q] \to [\alpha, \mathit{int}.\, \mathit{int} \to q]$. Applying, substituting and running it as before reads:

   $$\texttt{run}\ (\texttt{subst}\,\alpha_0\ (\mathbf{T}\,\alpha_0\ [\alpha_0.\, \mathit{square}])\ [\alpha_0.\, 2])\ 3$$

   which evaluates to 25.

Using the standard substitution properties of the three kinds of substitution, we can prove that this language is type-safe:

▶ **Theorem 8** (Type preservation)**.** *If* $M \downarrow V$ *then* $\cdot; \alpha_0 \vdash M : A$ *implies* $\cdot; \alpha_0 \vdash V : A$.

$$P, Q ::= x \mid \lambda x.\, P \mid P\, Q \mid [\hat{\Gamma}.\, P] \mid {\sim}P\{\tau\} \mid \Lambda\alpha.\, P \mid P\, \Gamma \qquad \text{Term}$$
$$\tau ::= \mathsf{id}_\alpha \mid \tau, P \qquad \text{Substitution}$$
$$\Sigma ::= \cdot \mid \Sigma; \Gamma \qquad \text{Environment Stack}$$

$\boxed{\Sigma \vdash P : A}$ Term P has type A in environment stack Σ

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x : A} \qquad \textsc{Lam}\ \frac{\Sigma; \Gamma, x : A \vdash P : B}{\Sigma; \Gamma \vdash \lambda x.\, P : A \to B} \qquad \textsc{App}\ \frac{\Sigma \vdash P : A \to B \qquad \Sigma \vdash Q : A}{\Sigma \vdash P\, Q : A \to B}$$

$$\textsc{Box}\ \frac{\Sigma; \Gamma \vdash P : A}{\Sigma \vdash [\hat{\Gamma}.\, P] : [\tilde{\Gamma}.\, A]} \qquad \textsc{Unbox}\ \frac{\Sigma \vdash P : [\tilde{\Psi}.\, A] \qquad \Sigma; \Gamma \vdash \tau : \tilde{\Psi}}{\Sigma; \Gamma \vdash {\sim}P\{\tau\} : A} \qquad \textsc{Gen}\ \frac{\Sigma \vdash P : A \qquad \alpha \notin \mathrm{FV}(\Sigma)}{\Sigma \vdash \Lambda\alpha.\, P : \forall\alpha.\, A}$$

$$\textsc{Inst}\ \frac{\Sigma \vdash P : \forall\alpha.\, A}{\Sigma \vdash P\, \Psi : \{\tilde{\Psi}/\alpha\}A}$$

$\boxed{\Sigma \vdash \tau : \tilde{\Psi}}$ Term P has type A in environment stack Σ

$$\textsc{Id}\ \frac{}{\Sigma; G(\alpha) \vdash \mathsf{id}_\alpha : \alpha} \qquad \textsc{Cons}\ \frac{\Sigma \vdash \tau : \tilde{\Psi} \qquad \Sigma \vdash P : A}{\Sigma \vdash (\tau, P) : (\tilde{\Psi}, A)}$$

Figure 3 Syntax and typing of λ_I^{ctx}, the implicit contextual λ-calculus

2.2 The implicit contextual λ-calculus λ_I^{ctx}

The examples above might seem more verbose than the discussion in the introduction. This is due in part to the let-box construct, which forces values to be defined outside of the box they are used in.[2] The language λ_I^{ctx} is an implicit version of λ^{ctx} that inlines the definitions of the let-box into a new *unbox* construct. The same idea was already proposed for $\lambda\Box$ [5].

The syntax of λ_I^{ctx} is shown on Fig. 3; it is identical to that of λ^{ctx}, except for the replacement of the two constructs $\mathsf{let}_\Box\, u = M\, \mathsf{in}\, N$ and $u\{\sigma\}$ by a unique *unbox* ${\sim}P\{\tau\}$. Informally, we go from the former to the latter by the textually substituting ${\sim}M$ for u in N. Now, the stage of an expression is not determined by the number of boxes traversed anymore, but by it minus the number of unboxes traversed. Therefore, it is necessary to maintain a stack of environments during typing; the top of the stack contains variables of the current stage, the deeper stack elements contain variables of earlier stages. The Box rule pops an environment from this stage[3], and the Unbox rule pushes one. In the latter, the substitution τ mediates between the one in the eliminated box and the one pushed. There is an embedding of λ_I^{ctx} into λ^{ctx}:

▶ **Theorem 9** (Explicit translation). *If* $\Sigma; \Gamma \vdash P : A$ *then there is* Δ, M *such that* $\Delta; \Gamma \vdash M : A$.

[2] It is also due to its Church-style syntax, with explicit generalization, instantiation and substitution. A practical language would use type inference.

[3] Operationally, it is reconstructed from the two halves $\hat{\Gamma}$ and $\tilde{\Gamma}$.

It is an easy generalization the existing proof for $\lambda\Box$ [5], which involves lifting unboxes out of the corresponding boxes as let-box. Note that nested unbox cause the introduction of let-box at different places, not necessarily at the closest enclosing box. For instance, $[\alpha.\, f\ [\beta.\, g\ {\sim}(h\ {\sim}x\{\mathsf{id}_\alpha\})\{\mathsf{id}_\beta\}]]$ is translated to $\mathsf{let}_\Box\, u = x\, \mathsf{in}\, [\alpha.\, f\ (\mathsf{let}_\Box\, v = h\ u\{\mathsf{id}_\alpha\}\, \mathsf{in}\, [\beta.\, g\ v\{\mathsf{id}_\beta\}])]$: the innermost unbox produces the outermost let-box, because it belongs to stage 0; conversely, the outermost unbox produces the innermost let-box because it belongs to stage 1.

3 Embedding environment classifiers

It is easy to see that λ^{ctx}, being based on contextual types, subsumes Davies and Pfenning's $\lambda\Box$ [5]: the type $\Box A$ of closed code corresponds to the box type $[\alpha.\, A]$ (for an arbitrary α). It is less immediate to see that it also subsumes $\lambda\bigcirc$ [4], in which code is allowed to be open. This is what we show in this section, using a generalization of $\lambda\bigcirc$ allowing safe code evaluation: Taha and Nielsen's λ^α [15].

3.1 Environment classifiers

Another design direction of type system for staged computations is to start from a simpy-typed λ-calculus and directly add in the *quote* and *unquote* constructs of, e.g., Lisp (we write them respectively $\langle E\rangle$ and ${\sim}E$). Their typing should enforce *binding-time correctness*: a computation should only depend on the result of computations in past or present stages. Syntactically, it translates as the property of *staged lexical scoping*: a variable is in scope only at the same stage as it was introduced.[4] It can be enforced by indexing the typing judgment with an integer n denoting the current stage; variable introduction saves n in the environment, and variable lookup takes it into account. Now, quote $\langle E\rangle$ and unquote ${\sim}E$ respectively increment and decrement this index, while introducing and eliminating the type $\langle A\rangle$ of code. This system, called $\lambda\bigcirc$ by Davies [4], is logically grounded: the type $\langle A\rangle$ corresponds to the "next" modality $\bigcirc A$ of linear-time temporal logic (LTL). However, nothing guarantees code to be closed like in the previous section; adding a `run` primitive of type $\bigcirc A \to A$ could fail trying to evaluate a variable with no value. For instance, reducing $\langle\lambda x.\, {\sim}(\texttt{run}\,\langle x\rangle)\rangle$, would try to evaluate x. This fact led Taha and Nielsen [15] to devise λ^α, that restricted evaluation by annotating expressions with the "boundaries" of each stage.

Syntactically (Fig. 4), they turned the integer index of $\lambda\bigcirc$ into a list X of identifiers, called *environment classifiers* $\alpha, \alpha_1, \alpha_2$ (let us reuse these names) and denoting the current stage; each variable in the environment is now annotated with such a string. In rule Lam, the current classifier string is recorded in the environment; rule Var applies only if the current classifier string matches the one in the environment. Type $\langle T\rangle^\alpha$ denotes the type of code at a stage α; it is introduced by quote $\langle E\rangle^\alpha$ and eliminated by unquote ${\sim}E$: Quote and Unquote respectively push and pop the classifier α off the classifier string X. The construct $\Lambda\alpha.\, E$ (let us reuse this notation) sets the boundary for stage α: it guarantees that α has not been used in a super-expression of E; the second premise of Gen ensures this. $\Lambda\alpha.\, E$ introduces the type $\forall\alpha.\, A$, which is eliminated by its dual $E\,\alpha$ (again, overloading notations from previous section)This way, a value of type $\forall\alpha.\, \langle T\rangle^\alpha$ is necessarily a closed piece of code, that can be safely evaluated. For instance, expression $\langle\lambda x.\, {\sim}(\Lambda\alpha.\, \langle x\rangle^\alpha)\rangle^\alpha$ is justly rejected.

[4] It could be in scope in previous stages, a feature known as *cross-stage persistence* that we defer to future work.

$$
\begin{array}{rll}
T, U & ::= p \mid T \to U \mid \langle T \rangle^{\alpha} \mid \forall \alpha.\, T & \text{Type} \\
E, F & ::= x \mid \lambda x.\, E \mid E\, F \mid \langle E \rangle^{\alpha} \mid {\sim}E \mid \Lambda \alpha.\, E \mid E\, \alpha & \text{Term} \\
\Xi & ::= \cdot \mid \Xi, x :^{X} T & \text{Environment} \\
X, Y & ::= \cdot \mid X\alpha & \text{Classifier string}
\end{array}
$$

$\boxed{\Xi \vdash^{X} E : T}$ Term E has type to T in X environment Ξ at stage X

$$
\textsc{Var}\ \frac{(x :^{X} T) \in \Xi}{\Xi \vdash^{X} x : T} \qquad
\textsc{Lam}\ \frac{\Xi, x :^{X} T \vdash^{X} E : U}{\Xi \vdash^{X} \lambda x.\, E : T \to U} \qquad
\textsc{App}\ \frac{\Xi \vdash^{X} E : T \to U \qquad \Xi \vdash^{X} F : T}{\Xi \vdash^{X} E\, F : T \to U}
$$

$$
\textsc{Quote}\ \frac{\Xi \vdash^{X\alpha} E : T}{\Xi \vdash^{X} \langle E \rangle^{\alpha} : \langle T \rangle^{\alpha}} \qquad
\textsc{Unquote}\ \frac{\Xi \vdash^{X} E : \langle T \rangle^{\alpha}}{\Xi \vdash^{X\alpha} {\sim}E : T} \qquad
\textsc{Gen}\ \frac{\Xi \vdash^{X} E : T \qquad \alpha \notin \mathrm{FV}(\Xi, X)}{\Xi \vdash^{X} \Lambda \alpha.\, E : \forall \alpha.\, T} \qquad
\textsc{Inst}\ \frac{\Xi \vdash^{X} E : \forall \alpha.\, T}{\Xi \vdash^{X} E\, \alpha : T}
$$

Figure 4 Syntax and typing of λ^{α}, also known as Environment classifiers

The operational semantics of λ^{α} can be found in Taha and Nielsen [15]. Most notably, the syntax of values cannot be captured by context-free grammar: they are indexed by n, the stage to which it belongs. There are two evaluation rule for each syntactic constructs: one for the case where $n = 0$, i.e., we should evaluate the expression, and one for the case where $n > 0$, i.e. the expression is code, and therefore not evaluated. As for λ^{ctx}, λ^{α} can safely be extended with a `run` primitive, of type $(\forall \alpha.\, \langle T \rangle^{\alpha}) \to \forall \alpha.\, T$, for the reasons evoked above. The addition of environment classifiers and quantification has no logical counterpart in LTL.

3.2 Going contextual: from λ^{α} to λ^{ctx}_{I}

We now turn to our main technical result: the translation from λ^{α} to λ^{ctx}_{I}, which turns quotes into boxes. This involves inferring some environment information missing from λ^{α}. For instance, the term:

$$\Lambda \alpha.\, \lambda f.\, \langle \lambda x.\, {\sim}(f\, \langle x \rangle^{\alpha}) \rangle^{\alpha} : \forall \alpha.\, (\langle p \rangle^{\alpha} \to \langle q \rangle^{\alpha}) \to \langle p \to q \rangle^{\alpha}$$

can be translated to:

$$\Lambda \alpha.\, \lambda f.\, [\alpha.\, \lambda x.\, {\sim}(f\, [\alpha, x.\, x])\{\mathsf{id}_{\alpha}, x\}] : \forall \alpha.\, ([\alpha, p.\, p] \to [\alpha, p.\, q]) \to [\alpha.\, p \to q]$$

which contains more information (look at the environments in boxes, both in terms and in types). In other words, we need to annotate terms with the environment in which they are well-typed, and convey this information. Our translation has three noteworthy characteristics:

- first, it does not strictly preserve types, since λ^{α} and λ^{ctx}_{I} have different grammars of types. They are however translated in a "reasonable" way (preserving their structure).
- secondly, missing information in λ^{α} is retrieved from typing; consequently, the translation takes a source typing derivation and produces a target typing derivation.
- lastly, it is not unique: a well-typed λ^{α} term corresponds potentially to a family of λ^{ctx}_{I} term and types; we choose to express these families with schemas of term and types, i.e., the output will contain "logic" variables, standing for environment contexts, together with constraints on their values.

Translation definition

We first introduce an infinite set of *logic variables* g_1, g_2, ... standing for environment contexts G (as in Fig. 1). The syntax of λ_I^{ctx} of Fig. 3 is extended with the new construct:

$$\Gamma ::= \ldots \mid g(\Gamma)$$

A logic variable g comes with an "prefix" environment Γ that will be substituted for the hole of the environment context G it stands for. Note that no typing rule is associated to this new construct; we will only consider well-typed environments containing no logic variables.

▶ **Definition 10** (Valuation)**.** A *valuation* ρ maps logic variables to environment contexts:

$$\rho ::= \cdot \mid \rho,\, g \mapsto G$$

Concatenation of valuations $\rho_1 @ \rho_2$ is defined only if $\mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset$.

▶ **Definition 11** (Instantiation)**.** Let ρ be a valuation and Γ an environment (resp. A a type, P a term, Σ a stack). We define the *instantiation* $\Gamma\rho$ (resp. $A\rho$, $P\rho$, $\Sigma\rho$) as the substitution of all logic variables in Γ (resp. A, P, Σ) by their value in ρ: (omitted cases are homomorphic)

$$\alpha\rho = \alpha \qquad (g(\Gamma))\rho = g(\Gamma\rho) \quad \text{if } (g \mapsto G) \notin \rho \tag{20}$$

$$(g(\Gamma))\rho = G(\Gamma\rho) \quad \text{if } (g \mapsto G) \in \rho \tag{21}$$

This definition is extended homomorphically to propositions and to typing judgments.

Note that in eq. (21), the substitution of an environment context variable $g(\Gamma_1)$ by its value involves substituting its hole with the associated environment value $\Gamma\rho$.

In the following, we suppose that an implicit set of used logic variables is maintained (a `gensym`). A logic variable is *fresh* if it is not contained in this set; each fresh variable used in an expression is added to this set. Now, the following function relates types in λ^α and λ_I^{ctx}:

▶ **Definition 12** (Type translation)**.** We define the function $[\![T]\!] = A$ as follows:

$$[\![p]\!] = p \qquad [\![T \to U]\!] = [\![T]\!] \to [\![U]\!]$$

$$[\![\forall\alpha.\, T]\!] = \forall\alpha.\, [\![T]\!] \qquad [\![\langle T\rangle^\alpha]\!] = [g(\alpha).\, [\![T]\!]] \quad g \text{ fresh}$$

Quotes are translated to boxes; the environment attached is left unspecified, except for the fact that it must have bottom variable α; for this purpose, we introduce the fresh logic variable g. In other words, this function $[\![\cdot]\!]$, maps a λ^α type to a family of λ_I^{ctx} types indexed by a set of environment contexts. The following helper function relates a λ^α environment Ξ to a λ_I^{ctx} environment Ψ at a certain stage X:

▶ **Definition 13** (Environment restriction)**.** We define function $\Xi|_X^\alpha = \Psi$ by induction on Ξ:

$$(\Xi, x :^X T)|_X^\alpha = \Xi|_X^\alpha, x : [\![T]\!] \qquad (\Xi, x :^Y T)|_X^\alpha = \Xi|_X^\alpha \text{ if } X \neq Y \qquad (\cdot)|_X^\alpha = \alpha \tag{22}$$

The result environment Ψ contains only the declarations of Ξ declared at stage X, associated with their translated type. Variable α will be put at the bottom of the resulting environment.

We now translate λ^α environments into equivalent λ_I^{ctx} environment stacks, restricted to a classifier string. We reshuffle its declarations from a style where all declarations carry their classifier string, to a style where they are grouped in stack cells according to their stages.

$$\textsc{Var}\ \frac{[\![\Xi]\!]_X = \Sigma;\Gamma \qquad x : A \in \Gamma}{[\![\Xi \vdash^X x : T]\!] = \Sigma;\Gamma \vdash x : A \ /\ \top}$$

$$\textsc{Lam}\ \frac{[\![\Xi, x :^X T \vdash^X E : U]\!] = \Sigma;\Gamma, x : A \vdash P : B \ /\ C}{[\![\Xi \vdash^X \lambda x.\, E : T \to U]\!] = \Sigma;\Gamma \vdash \lambda x.\, P : A \to B \ /\ C}$$

$$\textsc{App}\ \frac{[\![\Xi \vdash^X E : T \to U]\!] = \Sigma \vdash P : A \to B \ /\ C \qquad [\![\Xi \vdash^X F : T]\!] = \Sigma' \vdash Q : A' \ /\ C'}{[\![\Xi \vdash^X E\,F : U]\!] = \Sigma \vdash P\,Q : B \ /\ C \wedge C' \wedge A = A' \wedge \Sigma = \Sigma'}$$

$$\textsc{Box}\ \frac{[\![\Xi \vdash^{X\alpha} E : T]\!] = \Sigma;\Gamma \vdash P : A \ /\ C}{[\![\Xi \vdash^X \langle E\rangle^\alpha : \langle T\rangle^\alpha]\!] = \Sigma \vdash [\hat{\Gamma}.\, P] : [\tilde{\Gamma}.\, A] \ /\ C}$$

$$\textsc{Unbox}\ \frac{[\![\Xi \vdash^X E : \langle T\rangle^\alpha]\!] = \Sigma \vdash P : [\tilde{G}(\alpha).\, A] \ /\ C \qquad \Xi|^\alpha_{X\alpha} = \Gamma}{[\![\Xi \vdash^{X\alpha} {\sim}E : T]\!] = \Sigma;\Gamma \vdash {\sim}P\{\mathsf{id}(\hat{\Gamma})\} : A \ /\ C \wedge G(\alpha) = \Gamma}$$

$$\textsc{Gen}\ \frac{[\![\Xi \vdash^X E : T]\!] = \Sigma \vdash P : A \ /\ C}{[\![\Xi \vdash^X \Lambda\alpha.\, E : \forall\alpha.\, T]\!] = \Sigma \vdash \Lambda\alpha.\, P : \forall\alpha.\, A \ /\ \forall\alpha.\, C}$$

$$\textsc{Inst}\ \frac{[\![\Xi \vdash^X E : \forall\alpha.\, T]\!] = \Sigma \vdash P : \forall\alpha.\, A \ /\ C \qquad g \text{ fresh}}{[\![\Xi \vdash^X E\,\alpha : T]\!] = \Sigma \vdash P\,g(\alpha) : \{g(\alpha)/\alpha\}A \ /\ C}$$

Figure 5 Translation of typing derivations from λ^α to λ^{ctx}_I

▶ **Definition 14** (Environment translation). $[\![\Xi]\!]_X = \Sigma;\Gamma$ is defined by induction on X:

$$[\![\Xi]\!]_\cdot = \Xi|^{\alpha_0}_\cdot \qquad\qquad [\![\Xi]\!]_{X\alpha} = [\![\Xi]\!]_X; \Xi|^\alpha_{X\alpha} \tag{23}$$

Translating Ξ at classifier string X results in a stack Σ of $|X|$ environments. In the first equation, α_0 (the top-level environment variable) corresponds to the empty classifier string. In the second, we concatenate all declarations at a stage, putting its name α at the bottom.

▶ **Example 15.** $[\![f :^\cdot \langle p \to q\rangle^\alpha, x :^\alpha p, y :^\alpha q, z :^{\alpha\beta\gamma} p]\!]_{\alpha\beta} = \alpha_0, f : [g_1(\alpha).\, p \to q]; \alpha, x : p, y : q; \beta.$

We can now define the translation as a relation between derivations:

▶ **Definition 16** (Derivation translation). The judgment $[\![\Xi \vdash^X E : T]\!] = \Sigma \vdash P : A \ /\ C$ is defined by the rules of Fig. 5. It reads: the λ^α derivation $\Xi \vdash^X E : T$ is translated to a λ^{ctx}_I derivation $\Sigma \vdash P : A$ that possibly contains logic variables which must satisfy constraints C. Constraints are clauses in first-order predicate logic with equality.

The rules closely follow the typing rules of the two languages, projecting each λ^α construct to its λ^{ctx}_I equivalent, and (un-)quote to (un-)box. The need for a translation on typing derivations, and not only on terms and types, is witnessed by Box rule: the residual term $[\hat{\Gamma}.\, P]$ and its type contain Γ, that is reconstructed through typing. At the leaves

(rule Var), we translate the environment using Definition 14, potentially introducing logic variables in types. Rule Inst also introduces a logic variable g. Throughout the translation, constraints C are accumulated; they force the ultimate instantiation of logic variables. Var generates the vacuously true constraint. As for ML type inference, rule App constrains the argument's type A' and the function's domain A to be equal, as well as both premisses' environments. The Unbox rule inspects its subderivation's target type, which must be a box in an environment context G; we constrain this target environment to be the restriction of the source environment at the current stage. Finally, in the constraints generated above a Gen rule, variable α can be free; thus, we enclose it with a universal quantifier.

▶ **Example 17.** The following judgment is derivable:

$$\begin{aligned}&[\![\vdash^{\cdot}\ \lambda f.\,\Lambda\alpha.\,\langle\lambda x.\sim(f\,\alpha\,\langle x\rangle^{\alpha})\rangle^{\alpha} : (\forall\alpha.\,\langle p\rangle^{\alpha}\to\langle q\rangle^{\alpha})\to\forall\alpha.\,\langle p\to q\rangle^{\alpha}]\!] = \\ &\qquad\qquad \alpha_0 \vdash \lambda f.\,\Lambda\alpha.\,[\alpha.\,\lambda x.\sim(f\,(g_3(\alpha))\,[\alpha, x.\,x])\{\mathsf{id}_\alpha, x\}] : \\ &(\forall\alpha.\,[g_1(\alpha).\,p]\to[g_2(\alpha).\,q])\to\forall\alpha.\,[\alpha.\,p\to q]\ /\\ &\qquad\qquad (g_1(g_3(\alpha)) = \alpha, x : p)\wedge(g_2(g_3(\alpha)) = \alpha, x : p)\end{aligned}$$

These constraints accept several solutions, among which:

- $g_1 = g_2 = [], g_3 = [], x : p$, which instantiates the residual terms and types to:

$$\lambda f.\,\Lambda\alpha.\,[\alpha.\,\lambda x.\sim(f\,(\alpha, x : p)\,[\alpha, x.\,x])\{\mathsf{id}_\alpha, x\}] : (\forall\alpha.\,[\alpha.\,p]\to[\alpha.\,q])\to\forall\alpha.\,[\alpha.\,p\to q]$$

- $g_3 = [], g_1 = g_2 = [], x : p$, which gives the less general, alternative translation:

$$\lambda f.\,\Lambda\alpha.\,[\alpha.\,\lambda x.\sim(f\,\alpha\,[\alpha, x.\,x])\{\mathsf{id}_\alpha, x\}] : (\forall\alpha.\,[\alpha, p.\,p]\to[\alpha, p.\,q])\to\forall\alpha.\,[\alpha.\,p\to q]$$

Translation correctness

The main results are stated below. First, we can prove that the translation produces only correct derivations, provided that we instantiate the logic variables according to the constraints:

▶ **Theorem 18** (Type soundness). *If* $[\![\Xi\vdash^X E : T]\!] = \Sigma\vdash P : A\ /\ C$ *and* $C\rho$ *holds for some instantiation* ρ*, then* $(\Sigma\vdash P : A)\rho$*.*

The above statement would vacuously hold for a bogus translation mapping, e.g., all terms to a trivial derivation. It is not the case: the types and environments are translated according to Definitions 12 and 14, i.e., they only differ by the annotations on boxes.

▶ **Theorem 19** (Correctness). *If* $\Xi\vdash^X E : T$ *and* $[\![\Xi\vdash^X E : T]\!] = \Sigma\vdash P : A\ /\ C$ *then there exists* ρ *such that* $[\![T]\!]\rho = A$ *and* $[\![\Xi]\!]_X\rho = \Sigma$*.*

Finally, we can prove that these rules can be read as an algorithm, taking as input the source judgment and returning the target judgment and the constraints (the proof is constructive, which amounts to a decidability result). Simultaneously, we can show that the emitted constraints can always be solved:

▶ **Theorem 20** (Decidability). *If* $\Xi\vdash^X E : T$ *then there exists* Γ*,* P*,* A*,* C *and* ρ *such that* $[\![\Xi\vdash^X E : T]\!] = \Gamma\vdash P : A\ /\ C$ *and* $C\rho$ *holds.*

4 Related Work

According to Tim Sheard's taxonomy of meta-programming [14], the system we study is a homogeneous, multi-stage, statically typed, manual, run-time program generator using a quasi-quote representation. Most general-purpose programming languages feature some meta-programming facility: pre-processors, constant- or macro-definitions, template systems, staging etc. They serve two main purposes: performance (partial evaluation, compile-time optimization), and practical expressiveness (user-defined syntactic constructs). "Hygienic" (i.e., with lexical scoping) and well-typed macro-definitions systems have been studied extensively [6]. The present work stems from Davies and Pfenning's Curry-Howard correspondences between such type systems and two modal logics [4, 5]. We aimed at reconciling their two orthogonal systems, $\lambda\Box$ and $\lambda\bigcirc$, while retaining the clear logical correspondence.

Environment classifiers and the λ^α language [15] had a certain impact in making evaluation possible in a $\lambda\bigcirc$-like language. It was implemented as MetaML [16], then MetaOCaml and BER MetaOCaml [8]. Our language λ^{ctx} is strictly more expressive, thanks to its control over single free variables; we conjecture that it makes simpler to define macros which bind variables (in MacroML [6], binding notations are interpreted as functions). Tsukada and Igarashi studied the logical foundation of environment classifiers [17]; we refined their analysis, showing that they range over environments in a contextual logic.

The λ_{open} family of languages [7] are typed multi-staged ML-like derived from $\lambda\Box$ and featuring mutable state, cross-stage persistence, and two substitutions: one capture-avoiding and one intentionally not. It has a type of open code $\Box(\Gamma \triangleright A)$ interestingly similar to ours, and a notion of environment polymorphism; the authors, however, treat Γ as a row variable and use record subtyping for type inference. Its operational semantics is also more complex: it needs a `gensym` and non context-free values; the logical foundation is also not discussed. For comparison, it would be interesting to reformulate λ_{open} as an explicit, two-zone system. We also plan to investigate the logical interpretation of cross-stage presistence.

Recently, Rhiger proposed a type system for a multi-staged λ-calculus with (anti-)quotations featuring safe evaluation, open code manipulation and mutable state [13]. Interestingly, it also shares syntactic features with our language λ_I^{ctx}: a "contextual" type of code $[\gamma]t$ parametrized over an environment γ, and environment stacks in the typing judgment. It is however very different in essence: future-stage environments are kept when unquoting and not discarded as in λ_I^{ctx}, directly allowing open code. In this sense, it is closer to $\lambda\bigcirc$, but with a tracking of free variables which makes it less expressive: for instance, the two-level η-expansion is not well-typed in this system.

Contextual modal type theory [9] was designed to provide a principled foundation to the manipulation of open terms, as performed in proof and programming languages. From this work emerged Beluga, [12], a programming and reasoning language which allows to manipulate expressions with binders typed in the LF logical framework. Contrarily to this work, it uses only two, heterogeneous levels; it also features pattern-matching on data, and has a dependent type discipline. Pattern-matching on code could be added to our proposal, and would be novel and useful to design, e.g., program transformers or optimizers.

5 Conclusion

We have shown that a logically principled core λ-calculus, that we called λ^{ctx}, suffices to express safe multi-staged computations. Its instrumental features are contextual types and first-class environments. Contrarily to previous work [15, 13], it has a two-zone type system (validity and truth), a simple operational semantics with a context-free grammar of values,

and provides a fine-grained control over free variables, on a per-variable basis. In particular, it subsumes Taha and Nielsen's λ^α: we have shown a novel, type-preserving embedding from it, that infers missing information. Beyond expressiveness, we have shown that environment classifiers range over concrete environment contexts, giving a logical foundation to λ^α. As a consequence, we have also exhibited a new relationship between S4's necessity and LTL's "next" modalities: there is a necessity-based presentation of LTL, provided that we have quantification over environments.

Acknowledgments

The author would like to thank Brigitte Pientka for impulsing and guiding this work, and Beniamino Accatoli for the script-doctoring.

References

1 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In John Field and Michael Hicks, editors, *POPL 2012*, pages 413–424. ACM, 2012.
2 Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. *JFP*, 15(5):797–835, 2005.
3 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, 1996.
4 Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS 1996*, pages 184–195, 1996.
5 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.
6 Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ICFP 2001*, pages 74–85, 2001.
7 Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL 2006*, pages 257–268, 2006.
8 Oleg Kiselyov. The design and implementation of BER MetaOCaml - system description. In *FLOPS 2014*, pages 86–102, 2014.
9 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.
10 Flemming Nielson and Hanne Riis Nielson. *Two-level Functional Languages*. Cambridge University Press, New York, NY, USA, 1992.
11 Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL 2008*, pages 371–382, 2008.
12 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR 2010*, pages 15–21, 2010.
13 Morten Rhiger. Staged computation with staged lexical scope. In *ESOP 2012*, pages 559–578, 2012.
14 Tim Sheard. A taxonomy of meta-programming systems, 2015. Accessed: 2015-02-15.
15 Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL 2003*, pages 26–37, 2003.
16 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
17 Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. *Logical Methods in Computer Science*, 6(4), 2010.