Closing the Stage

From Staged Code to Typed Closures

Yukiyoshi Kameyama
University of Tsukuba
kameyama@acm.org

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Abstract

Code generation lets us write well-abstracted programs without performance penalty. Writing a correct code generator is easier than building a full-scale compiler but still hard. Typed multistage languages such as MetaOCaml help in two ways: they provide simple annotations to express code generation, and they assure that the generated code is well-typed and well-scoped. Unfortunately, the assurance only holds without side effects such as state and control. Without effects, generators often have to be written in a continuation-passing or monadic style that has proved inconvenient. It is thus a pressing open problem to combine effects with staging in a sound type system.

This paper takes a first step towards solving the problem, by translating the staging away. Our source language models MetaOCaml restricted to one future stage. It is a call-by-value language, with a sound type system and a small-step operational semantics, that supports building open code, running closed code, cross-stage persistence, and non-termination effects. We translate each typing derivation from this source language to the unstaged System F with constants. Our translation represents future-stage code using closures, yet preserves the typing, α-equivalence (hygiene), and (we conjecture) termination and evaluation order of the staged program.

To decouple evaluation from scope (a defining characteristic of staging), our translation weakens the typing environment of open code using a term coercion reminiscent of Gödel's translation from intuitionistic to modal logic. By converting open code to closures with typed environments, our translation establishes a framework in which to study staging with effects and to prototype staged languages. It already makes scope extrusion a type error.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; polymorphism; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Design, Languages

Keywords Multistage programming, type abstraction, parametric polymorphism, mutable state and control effects, closures

PEPM'08, January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001...$5.00

1. Introduction

Code generation is the most promising approach in high-performance computing (Püschel et al. 2005) and high-assurance embedded programming (Hammond and Michaelson 2003). Because a generic program in an earlier stage can generate a specialized program in a later stage and assure it safe, the programmer need not trade off abstraction and assurance for efficiency in time and space.

An attractive way to express such code generation is to use a *multistage programming language* such as MetaOCaml (Lengauer and Taha 2006), in which the programmer need only annotate the program to separate the stages (Taha 2004). Staging provides a principled interface through which a code generator can take advantage of the implementation of the language it is written in, whether or not that implementation is a self-interpreter that can be specialized (Jones 1988). The applications of staging thus include the usual uses of code generation, such as partial evaluation (Davies and Pfenning 2001; Ghani et al. 1998), embedding domain-specific languages (Czarnecki et al. 2004; Pašalić et al. 2002), and controlling special processors (Elliott 2004; Taha 2005).

Staging lets us symbolically manipulate pieces of *open code*, which may contain free variables that will be bound to actual values only at a future stage. Yet a staged language should ensure that α-conversion preserves the meaning of a program, and that the code finally generated is closed, that is, does not contain free variables (Taha and Nielsen 2003). Staged type systems exist in the literature that achieve these goals (Calcagno et al. 2004; Taha and Nielsen 2003), but they do not consider side effects with open code.

Side effects let us apply code-generation techniques such as let insertion without cumbersome programming in continuation-passing or monadic style. In particular, delimited control is useful for partially evaluating programs using sum types (Balat et al. 2004; Lawall and Danvy 1994) and delimited control (Asai 2002), and mutable state and delimited control make it easy to express let insertion (Sumii and Kobayashi 2001; Swadi et al. 2006) and to count generated operations. Unfortunately, these effects with open code make it possible to generate ill-scoped code—that is, to cause *scope extrusion*—in all multistage type systems today.

Our motivation is to write effectful code generators in a language like MetaOCaml without risking scope extrusion. In other words, we want a multistage type system that is sound with effects.

Contributions To design a multistage type system that is sound with effects, we must understand how effects interact with staging. To this end, we translate in this paper a language λ_{1v}^{α}, which models the mature and practical multistage language MetaOCaml (§3), into System F, which lacks staging (§4). Our translation incurs the overhead of functional and type abstraction, but keeps the speedup of specialization and expresses the static and (we conjecture) dynamic semantics of λ_{1v}^{α}, including the order of effects among stages (§5). Our translation reduces both *closed* and *open* code to closures.

Because we aim to combine staging with effects (especially delimited control), we formalize our source language using a small-step operational semantics with evaluation contexts (Wright and Felleisen 1994). With staging, our redexes may be open and evaluation contexts may be binding (§3.2). Such a small-step semantics has only been used to study call-by-need (Ariola and Felleisen 1997; Maraist et al. 1998) to our knowledge. We prove subject reduction, determinism, and progress for this source language (§3.3).

After describing the translation, we present three payoffs: strong normalization for the source language without fix (§5.3), detecting scope extrusion (§6), and counting arithmetic operations in a sound staged language (§6).

MetaOCaml supports an unlimited number of future stages (Glück and Jørgensen 1997), so generated code may in turn generate code. In this paper, we restrict ourselves to one future stage, which suffices in many real-life applications (Carette and Kiselyov 2005; Lengauer and Taha 2006) and where issues such as scope extrusion and cross-stage persistence already arise (see §2). These issues force us to type environments more precisely yet polymorphically, which complicates and distinguishes our translation as compared to typed closure conversion (Minamide et al. 1996).

2. The basics of staging and our translation

We begin by informally sketching our translation on some examples, starting with the classic power function. We use the language formalized in §3, which is like OCaml with staging annotations (explained below) and fix (in place of `let rec`). Our examples in MetaOCaml and translations in OCaml are all available online at `http://okmij.org/ftp/Computation/staging/`.

```
let $square = \lambda x{:}\mathsf{int}.\, x \times x$
let $power$ = (* $(\alpha)\,(\mathsf{int} \to \langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{int}\rangle^\alpha)$ *)
    $(\alpha)\,\mathsf{fix}\, f(n:\mathsf{int}):\langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{int}\rangle^\alpha.\ \lambda x{:}\langle\mathsf{int}\rangle^\alpha.$
    if $n = 0$ then $\langle 1\rangle^\alpha$
              else if $n \bmod 2 = 0$ then $\langle \%square \sim(f\ (n/2)\ x)\rangle^\alpha$
                                         else $\langle \sim x \times \sim(f\ (n-1)\ x)\rangle^\alpha$
let $power7$ = (* $(\alpha)\,(\mathsf{int} \to \mathsf{int})$ *)
    $\mathsf{run}(\alpha)\langle \lambda x^\alpha{:}\mathsf{int}.\sim(printf$ “power”; $power[\alpha]\ 7\ \langle x^\alpha\rangle^\alpha)\rangle^\alpha$
let $res = (\alpha)\,(power7\,[\alpha]\,2,\, power7\,[\alpha]\,3)$
```

This code uses staging annotations: bracket $\langle e\rangle^\alpha$, escape $\sim e$, cross-stage persistence $\%e$, and $\mathsf{run}\, e$. The superscript α is a *classifier*, explained in §3. To ease the notation, we will often drop the classifier where it can be easily inferred according to the syntax of §3; for now, classifier annotations along with the classifier introduction $(\alpha)e$ and the classifier application $e[\alpha]$ may be disregarded. If we ignore these annotations—that is, regard $\langle e\rangle^\alpha$, $\sim e$, $\%e$, $\mathsf{run}\, e$, $(\alpha)e$, and $e[\alpha]$ as just e, and x^α as just x—then the code expresses a standard way to compute x^n. The expression

$$\lambda x{:}\mathsf{int}.\,(printf\ \text{“power”};\ power\ 7\ x)$$

builds a closure that includes the *printf* operation and the *power* computation. The body of the closure is of course not evaluated until the argument x is supplied, so nothing is printed when *power7* is computed. Computing *res* invokes *power7* twice and so prints “power” twice with the unstaged code.

With staging annotations, the picture is different. The code still expresses the same algorithm, but the computations are performed at different stages. The *bracket* $\langle e\rangle^\alpha$, like `quasiquote` in Lisp, says to execute e at a future stage. If e has the type `int`, then $\langle e\rangle^\alpha$ has the type $\langle\mathsf{int}\rangle^\alpha$. The *escape* $\sim e$, like `unquote` in Lisp, says to evaluate e while building a future-stage computation. The result of e at the present stage must be a future-stage computation and is spliced into the future-stage computation being built. The construct $\mathsf{run}\, e$, like `eval` in Lisp, evaluates the computation e.

With staging annotations in the code above, “power” is printed only once—when *power7* is evaluated. Consequently, when we compute *res*, nothing is printed, and the function *power* is not executed because the loop over n has been already performed. The body of *power7* is $\lambda x.\, x \times square\ (x \times square\ (x \times 1))$.

2.1 Naïve non-solution

Our goal is to represent staged computations in a regular language, namely System F (§4). The essence of staging is to quote computations, delaying their execution to some future time. In call-by-value languages, the standard way to delay a computation is to put it into a function or simply a thunk. We may be tempted then to represent the type $\langle t\rangle$ as $() \to t$, the bracket expression $\langle e\rangle$ as $\lambda().\,e$, the escape $\sim e$ and $\mathsf{run}\, e$ as $e\ ()$, and cross-stage persistence $\%e$ as e.

If we interpret our staged power code that way, it works just like the unstaged code: “power” is printed twice, when we compute *res*. Thus, we got rid of staging along with its benefit. This failure to represent staging can be understood by considering an expression $\langle\sim e\rangle$. The staged language evaluates e while building the bracketed expression, so any side effect in e will occur then. Our naïve interpretation turns the expression into $\lambda().\,e()$. Here e becomes encapsulated into a thunk, so it will be evaluated when the bracketed computation is run, not while it is built.

For the thunk to encapsulate only the result of an escape and not the escape itself, we need to ‘lift’ the escape out of the thunk, that is, interpret $\langle\sim e\rangle$ as $\mathsf{let}\ v = e\ \mathsf{in}\ \lambda().\,v()$. But then, the expression

$$\langle \lambda x^\alpha{:}\mathsf{int}.\sim(printf\ \text{“power”};\ power[\alpha]\ 7\ \langle x\rangle)\rangle$$

in *power7* above would become

$$\mathsf{let}\ v = printf\ \text{“power”};\ power[\alpha]\ 7\ \lambda().x\ \mathsf{in}\ \lambda().\,\lambda x{:}\mathsf{int}.\,v()$$

Now *printf* occurs outside the thunk, but so does the unbound variable x. After all, we are supposed to compute v before a value is supplied for x. Staging lets us use a variable symbolically before it is bound to a value. Our translation must thus deal with symbolic references to variables without values (Taha 1999; §7.2.1).

2.2 Environment passing

To translate symbolic variable references, we can model the environment as a record whose keys are such first-class references. To explore this idea, let us introduce an abstract data type of extensible records: $\mathfrak{R}_\circ$ is some initial environment; $\mathfrak{R}('x = v)$ extends the environment $\mathfrak{R}$ by associating the key $'x$ to the value v; and $\mathfrak{R}('x)$ looks up the value associated with the key $'x$ in the environment $\mathfrak{R}$.

We can represent a code expression $\langle e\rangle$ as $\lambda\mathfrak{R}.\ldots$, where the argument $\mathfrak{R}$ associates the free variables in e to their values. For example, we represent $\langle x\rangle$ as $\lambda\mathfrak{R}.\,\mathfrak{R}('x)$, a projection from the environment. This function is first-class and can be passed around before we have a concrete environment to apply it to. We represent $\mathsf{run}\, e$ by applying the representation of e to $\mathfrak{R}_\circ$. Likewise, we interpret the escape $\langle\sim\langle 1\rangle\rangle$ as $\mathsf{let}\ v = \lambda\mathfrak{R}.\,1\ \mathsf{in}\ \lambda\mathfrak{R}.\,v\mathfrak{R}$. A binding operation under bracket should extend the effective environment, so we can translate $\mathsf{let}\ f y = \langle\sim y + 1\rangle\ \mathsf{in}\ \langle\lambda x.\sim(f\langle x\rangle)\rangle$ as

$$\begin{aligned}&\mathsf{let}\ f y = \lambda\mathfrak{R}.\,y\mathfrak{R} + 1\ \mathsf{in}\\ &\qquad\qquad \mathsf{let}\ v = f(\lambda\mathfrak{R}.\,\mathfrak{R}('x))\ \mathsf{in}\ \lambda\mathfrak{R}.\,\lambda x.\,v\,(\mathfrak{R}('x = x)).\end{aligned}$$

For this translation to work, we must ensure that look-up always succeeds. We prove it does below by assigning types to environments that reflect their contents. We should also specify how to α-rename a translated expression: it is easy to rename x to y in $\langle\lambda x.\sim(f\langle x\rangle)\rangle$ without affecting the meaning of the expression, but it is more involved to perform the same renaming on our translation, because we need to replace x with y as well as $'x$ with $'y$. Before addressing these issues, we use this approach to translate *power*.

$$\begin{array}{l}
\mathsf{let}\ \mathit{square} = \lambda x.x \times x \\
\mathsf{let}\ \mathit{power} = \mathsf{fix}\, f(n).\ \lambda x. \\
\quad \text{if } n = 0 \text{ then } \lambda\mathfrak{R}.\,1 \\
\qquad\qquad \text{else if } n \bmod 2 = 0 \\
\qquad\qquad\quad \text{then } \mathsf{let}\ v = f\ (n/2)\ x\ \mathsf{in}\ \lambda\mathfrak{R}.\,\mathit{square}(v\mathfrak{R}) \\
\qquad\qquad\quad \text{else } \mathsf{let}\ v_1 = x\ \mathsf{in}\ \mathsf{let}\ v_2 = f\ (n-1)\ x\ \mathsf{in} \\
\qquad\qquad\qquad \lambda\mathfrak{R}.\,v_1\mathfrak{R} \times v_2\mathfrak{R} \\
\mathsf{let}\ \mathit{power7} = \mathsf{let}\ v = \mathit{printf}\ \text{“power”};\ \mathit{power}\ 7\ \lambda\mathfrak{R}.\,\mathfrak{R}('x)\ \mathsf{in} \\
\qquad\qquad (\lambda\mathfrak{R}.\,\lambda x.\,v\,(\mathfrak{R}('x = x)))\,\mathfrak{R}_\circ \\
\mathsf{let}\ \mathit{res} = (\mathit{power7}\ 2, \mathit{power7}\ 3)
\end{array}$$

As in the staged code, the string “power” is printed only once, when we compute *power7*.

2.3 Environment polymorphism

A common practice in staged programming is to splice a piece of open code into a scope with additional bindings. Translating this practice requires polymorphism, as we explain with a simple example. The example is a function *ef* from the code type $\langle \mathsf{int}\rangle^\alpha$ to the code type $\langle \mathsf{int} \to \mathsf{int}\rangle^\alpha$. It splices the argument into code with an extra binding. Below we define *ef* and use it in two examples.

$$\begin{array}{l}
\mathsf{let}\ \mathit{ef} = (\alpha)\lambda z{:}\langle\mathsf{int}\rangle^\alpha.\,\langle\lambda x^\alpha{:}\mathsf{int}.\,{\sim}z + x^\alpha\rangle^\alpha \\
\mathsf{let}\ \mathit{ef}_1 = (\alpha)\mathit{ef}[\alpha]\langle 1\rangle^\alpha \\
\mathsf{let}\ \mathit{ef}_2 = (\alpha)\langle\lambda x^\alpha.\,\lambda y^\alpha.\,{\sim}(\mathit{ef}[\alpha]\langle x^\alpha \times y^\alpha\rangle^\alpha)\rangle^\alpha
\end{array}$$

The term ef_1 evaluates to $(\alpha)\langle\lambda x.\,1+x\rangle$ whereas ef_2 evaluates to $(\alpha)\langle\lambda x.\,\lambda y.\,\lambda x'.x \times y + x'\rangle$. In the latter result, we need to distinguish two later-stage variables named x. To maintain hygiene and α-equivalence, we must lexically link each use of a variable to its binding occurrence and rename variables if their names clash.

Applying the informal translation of the previous section to this example shows several problems.

$$\begin{array}{l}
\mathsf{let}\ \mathit{ef} = \lambda z.\,\lambda\mathfrak{R}.\,\lambda x.\,z(\mathfrak{R}('x = x)) + x \\
\mathsf{let}\ \mathit{ef}_1 = \mathit{ef}(\lambda\mathfrak{R}.\,1) \\
\mathsf{let}\ \mathit{ef}_2 = \ \mathsf{let}\ v = \mathit{ef}(\lambda\mathfrak{R}.\,\mathfrak{R}('x) \times \mathfrak{R}('y))\ \mathsf{in} \\
\qquad\qquad \lambda\mathfrak{R}.\,\lambda x.\,\lambda y.\,v(\mathfrak{R}('x = x)('y = y))
\end{array}$$

The first problem is that *ef* needs a polymorphic type. To type-check the multiplication $\mathfrak{R}('x) \times \mathfrak{R}('y)$, the type of $\mathfrak{R}$ should associate the keys $'x$ and $'y$ with the type int, but $\mathfrak{R}$ in ef_1 may not contain a mapping for $'y$. In general, we may invoke *ef* in the scope of any number of later-stage variables, so we seem to need so-called ρ-polymorphism for our environment records. The use of *ef* should be typed in ef_1 as

$$\forall\rho.\,(\{\rho\} \to \mathsf{int}) \to (\{\rho\} \to \mathsf{int} \to \mathsf{int})$$

but in ef_2 as

$$\forall\rho.\,(\{'x{:}\mathsf{int}, 'y{:}\mathsf{int}, \rho\} \to \mathsf{int}) \to (\{'x{:}\mathsf{int}, 'y{:}\mathsf{int}, \rho\} \to \mathsf{int} \to \mathsf{int}).$$

The α-renaming problem noted above also rears its head. To maintain hygiene, the translation needs to detect potential name clashes among variables and avert them by renaming variables along with fields in $\mathfrak{R}$. The translated example above does not work because the extensions $\mathfrak{R}('x = x)$ on the first and last lines clash. To prevent such a clash, our type for $\mathfrak{R}$ seems to require not only ρ-polymorphism but also negative side conditions that state the names that must *not* occur in any instantiation of the polymorphic type. Such conditions complicate type checking, especially with multiple staging levels, yet still leave the problem of preserving the meaning of programs in the face of α-conversion (Taha and Nielsen 2003; §1.4). Taha and Nielsen avoid these difficulties by ingeniously introducing classifiers, which track just enough information about the structure of environments to ensure type safety in a staged language without effects. Alas, we need more precision to maintain hygiene in a language with effects or without staging support.

2.4 Maintaining hygiene

We maintain α-equivalence and avoid name clashes by representing $\mathfrak{R}$ not as a record keyed by names but as a tuple keyed by indices. The typing environment of the staged code is an ordered sequence of bindings, so we can use that ordering to index into $\mathfrak{R}$ at run time. This idea is reminiscent of de Bruijn indices, but we index only later-stage variables and pass the indices around during the earlier stage as first-class code values.

The translation of a code expression thus depends on the later-stage variables in scope. For example, the code expression $\langle 1\rangle$ translates to $\lambda().\,1$ in the empty typing environment. The expression $\langle x\rangle$ translates to $\lambda x.x$ if x is the only later-stage variable in the typing environment, but to $\lambda(x,y).x$ if the typing environment contains the later-stage variables x and y, in that order. To splice in a code value is to apply it to the current later-stage environment reified as a tuple, so the translation of $\langle\lambda x.\,\lambda y.\,{\sim}\langle x \times x\rangle + y \times y\rangle$ is essentially $\lambda().\,\lambda x.\,\lambda y.\,(\lambda(x,y).x \times x)(x,y) + y \times y$.

A code value may be created in one typing environment then used in many others, as in

$$\mathsf{let}\ z = \langle 1\rangle\ \mathsf{in}\ \langle\lambda x^\alpha{:}\mathsf{int}.\,{\sim}z + x^\alpha\rangle. \tag{1}$$

The code value z is created in an environment with no later-stage variable, then used in an environment with one later-stage variable x. In general, the environment of use must extend the environment of creation; it is no accident that this crucial invariant holds in the absence of effects. Having translated $\langle 1\rangle$ to $\lambda().\,1$ in the empty environment, we would commit a type error were we to translate ${\sim}z$ in (1) by applying $\lambda().\,1$ to (x). Instead, we need to *coerce* the code value to accommodate the extended environment. Given the environment of use and of creation, it is easy to see that we should apply the coercion $\lambda f.\,\lambda(x).\,f()$ to $\lambda().\,1$. We thus translate (1) to

$$\mathsf{let}\ z = \lambda().\,1\ \mathsf{in}\ \lambda().\,\lambda x.\,(\lambda f.\,\lambda(x).\,f())\ z\ (x) + x. \tag{2}$$

(Actually, our formal translation produces some additional β-value redexes. The gory details are shown in §5.2.)

A second sort of polymorphism arises when a function takes a code value as argument, as *ef* above does. The translation of the code argument is a function from environments, but we do not know those environments' type when translating the function because the code argument may use any number of later-stage variables. Therefore the function must translate to a polymorphic function whose type is of the form $\forall\pi.\,(\pi \to \cdots) \to \cdots$. For example, we translate *ef* to a function of type $\forall\pi.\,(\pi \to \mathsf{int}) \to (\pi \to \mathsf{int} \to \mathsf{int})$. As the examples using *ef* in §2.3 show, each application of *ef* may instantiate the type variable π to a different environment type. In particular, we translate ef_1 and ef_2 there as follows.

$$\begin{array}{l}
\mathsf{let}\ \mathit{ef}_1 = \mathit{ef}[()](\lambda().\,1) \\
\mathsf{let}\ \mathit{ef}_2 = \mathsf{let}\ v = \mathit{ef}[(\mathsf{int},\mathsf{int})](\lambda(x,y).x \times y)\ \mathsf{in}\ \lambda().\,\lambda x.\,\lambda y.\,v(x,y)
\end{array}$$

It remains to translate *ef* itself, or equivalently, to translate its body $\langle\lambda x^\alpha{:}\mathsf{int}.\,{\sim}z + x^\alpha\rangle^\alpha$ in the environment $z : \langle\mathsf{int}\rangle^\alpha$. Again because the code value z may use any number of later-stage variables, the translated type of z is $\pi \to \mathsf{int}$ where π is a Λ-bound type variable. The code value returned by the translation of *ef* should have the type $\pi \to \mathsf{int} \to \mathsf{int}$. Guided by these types and using the techniques described above, we obtain the translation

$$\mathsf{let}\ \mathit{ef} = \Lambda\pi.\ \lambda z{:}\pi \to \mathsf{int}.\,\lambda r{:}\pi.\,\lambda x{:}\mathsf{int}.\,(\lambda f.\,\lambda(r,x).\,fr)z(r,x) + x.$$

2.5 A higher-order example

A more complex example to translate is *staged η-expansion*, a higher-order function on code values that is useful in staged programming and difficult for staged type systems (Taha and Nielsen 2003; §1.4). Below we define *eta* and give an example of its use.

Classifiers	α, β	Types	$t ::= \mathsf{int} \mid t \to t \mid (\alpha)t \mid \langle u \rangle^\alpha$
Named levels	$A, B ::= 0 \mid \alpha$	Flat types	$u ::= \mathsf{int} \mid u \to u$
Variables	x^A, y^A, z^A, f^A	Environments	$\Gamma ::= [] \mid \Gamma, \alpha \mid \Gamma, x : t \mid \Gamma, x^\alpha : u$
Expressions	$e^0 ::= i^0 \mid x^0 \mid \lambda x^0{:}t.\, e^0 \mid \mathsf{fix} f^0(x^0 : t) : t.\, e^0 \mid e^0 + e^0 \mid e^0 e^0 \mid (\alpha)e^0 \mid e^0[\alpha] \mid \mathsf{run}\, e^0 \mid \langle e^\alpha \rangle^\alpha$		
	$e^\alpha ::= i^\alpha \mid x^\alpha \mid \lambda x^\alpha{:}u.\, e^\alpha \mid \mathsf{fix} f^\alpha(x^\alpha : u) : u.\, e^\alpha \mid e^\alpha + e^\alpha \mid e^\alpha e^\alpha \mid {\sim} e^0 \mid \%e^0$		
Values	$v^0 ::= i^0 \mid x^0 \mid \lambda x^0{:}t.\, e^0 \mid \mathsf{fix} f^0(x^0 : t) : t.\, e^0 \mid (\alpha)v^0 \mid \langle v^\alpha \rangle^\alpha$		
	$v^\alpha ::= i^\alpha \mid x^\alpha \mid \lambda x^\alpha{:}u.\, v^\alpha \mid \mathsf{fix} f^\alpha(x^\alpha : u) : u.\, v^\alpha \mid v^\alpha + v^\alpha \mid v^\alpha v^\alpha \mid \%e^0$		
Contexts	$C^0[\,] ::= [\,] \mid C^0[[\,] + e^0] \mid C^0[v^0 + [\,]] \mid C^0[[\,]e^0] \mid C^0[v^0[\,]] \mid C^0[(\alpha)[\,]] \mid C^0[[\,][\alpha]] \mid C^0[\mathsf{run}\,[\,]] \mid C^\alpha[{\sim}[\,]]$		
	$C^\alpha[\,] ::= C^\alpha[[\,] + e^\alpha] \mid C^\alpha[v^\alpha + [\,]] \mid C^\alpha[[\,]e^\alpha] \mid C^\alpha[v^\alpha[\,]] \mid C^\alpha[\lambda x^\alpha{:}u.\,[\,]] \mid C^\alpha[\mathsf{fix} f^\alpha(x^\alpha : u) : u.\,[\,]] \mid C^0[\langle[\,]\rangle^\alpha]$		

Figure 1. Syntax of λ_{1v}^α

$$\mathsf{let}\ eta = (\alpha)\lambda f : \langle \mathsf{int} \rangle^\alpha \to \langle \mathsf{bool} \rangle^\alpha .\, \langle \lambda x^\alpha{:}\mathsf{int}.\, {\sim}(f \langle x^\alpha \rangle^\alpha) \rangle^\alpha$$
$$\mathsf{let}\ eta_1 = (\alpha)\langle \lambda y^\alpha .\, \lambda u^\alpha .\, {\sim}(eta[\alpha](\lambda z.\, \langle {\sim}z < y^\alpha \times u^\alpha \rangle^\alpha)) \rangle^\alpha$$

The term eta_1 evaluates to $(\alpha)\langle \lambda y^\alpha .\, \lambda u^\alpha .\, \lambda x^\alpha .\, x^\alpha < y^\alpha \times u^\alpha \rangle^\alpha$.

The function f passed to *eta* maps the open code $\langle x^\alpha \rangle$ to code that may contain free variables other than x^α. In eta_1, for example, f splices $\langle x^\alpha \rangle$ into the open code $\langle x^\alpha < y^\alpha \times u^\alpha \rangle$, which uses the additional free variables y^α and u^α. Because f and *eta* introduce their free variables *separately*, they are each polymorphic in the part of the later-stage environment extended by the other (O'Hearn and Tennent 1995). That is, the translation of *eta* uses a type variable π to represent whatever free variables f introduces (namely y^α and u^α in eta_1), and the translation of f uses a type variable π' to represent whatever free variable *eta* introduces (namely x^α). The translation of *eta* thus has the polymorphic function type

$$\forall \pi.\, (\forall \pi'.\, ((\pi, \pi') \to \mathsf{int}) \to ((\pi, \pi') \to \mathsf{bool})) \to (\pi \to \mathsf{int} \to \mathsf{bool}), \quad (3)$$

in which π is to be instantiated by the caller and π' by the callee.

In particular, the caller eta_1 instantiates π to the tuple type $(\mathsf{int}, \mathsf{int})$ corresponding to y^α and u^α. Our formal translation performs this instantiation as part of coercing *eta* from its environment of creation, which is empty, to its environment of use in eta_1, which binds y^α and u^α. As formalized in §5.1, this coercion

$$\begin{aligned}&\lambda g.\, \lambda f.\, g[(\mathsf{int}, \mathsf{int})]\\ &\quad (\Lambda \pi'.\, \lambda z.\, \mathsf{let}\ z' = f[\pi'](\lambda(y, u, r).\, z((y, u), r))\ \mathsf{in}\\ &\qquad \lambda((y, u), r).\, z'(y, u, r))\end{aligned}$$

is a function from the type (3) to the type

$$(\forall \pi'.\, ((\mathsf{int}, \mathsf{int}, \pi') \to \mathsf{int}) \to ((\mathsf{int}, \mathsf{int}, \pi') \to \mathsf{bool})) \to ((\mathsf{int}, \mathsf{int}) \to \mathsf{int} \to \mathsf{bool}). \quad (4)$$

Our translation of eta_1 is then essentially

$$\begin{aligned}&\mathsf{let}\ v = eta[(\mathsf{int}, \mathsf{int})](\Lambda \pi'.\, \lambda z.\, \lambda((y, u), r).\, z((y, u), r) < y \times u)\ \mathsf{in}\\ &\lambda().\, \lambda y.\, \lambda u.\, v(y, u).\end{aligned}$$

Meanwhile, it is easier to translate *eta* to the type (3), as follows.

$$\mathsf{let}\ eta = \Lambda \pi.\, \lambda f.\, \mathsf{let}\ v = f[\mathsf{int}](\lambda(r, x).\, x)\ \mathsf{in}\ \lambda r.\, \lambda x.\, v(r, x)$$

3. The source language λ_{1v}^α

Figure 1 presents the syntax of our source language λ_{1v}^α, a simply-typed call-by-value λ-calculus with fix and staging annotations. This calculus is closely based on Taha and Nielsen's λ^α (2003) but limited to only one future stage. Modulo this limitation, the two calculi are equally expressive, as detailed in §3.4.

As our metavariable notation indicates, we superscript each expression e^A, value v^A, variable x^A, and context $C^A[\,]$ with a *named level A*, to be explained shortly. Literal constants, variables, abstractions, applications, and additions are standard. We define just one base type, int, but our examples use bool analogously. Because our language is call-by-value, we write an η-expanded fixpoint $\mathsf{fix} f^A(x^A : t_1) : t_2.\, e$ for a recursive function with the argument type t_1 and return type t_2, whose body e may refer to the function f as well as the argument x. The bound variables are annotated with their types, although we will often omit the types when they can be inferred. The form $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ abbreviates $(\lambda x.\, e_2)e_1$ as usual.

The language also contains brackets $\langle e \rangle^\alpha$, escapes ${\sim}e$, and $\mathsf{run}\, e$. There is also an explicit form $\%x$ for *cross-stage persistence* (CSP), left implicit in MetaOCaml. Bracket expressions $\langle e \rangle^\alpha$ and code types $\langle t \rangle^\alpha$ are labeled by an *environment classifier* α (Taha and Nielsen 2003). Classifiers are distinct identifiers that associate code expressions with the environments in which they are typed. A classifier α is bound by a so-called α-closed type expression $(\alpha)t$, which is akin to $\forall \alpha.t$ universally quantifying over a phantom type (Launchbury and Peyton Jones 1995) or a nonce name (Miller and Tiu 2003). An α-closed type is introduced by a classifier generalization expression $(\alpha)e$ and eliminated by a classifier instantiation expression $e[\alpha]$. A classifier generalization expression $(\alpha)e$ asserts that the classifier α is used exclusively in e; that is, it introduces a fresh new classifier α to label code in e, akin to $\Lambda\alpha.\, e$ generalizing over a type variable α. Dually, a classifier instantiation expression $e[\alpha]$ is akin to type application; it instantiates an α-closed expression with a particular classifier. The type system uses classifiers to prevent running open code.

Named levels A (or *levels* for short) generalize numbered levels in staged calculi (Glück and Jørgensen 1997; Nielson and Nielson 1996). A level is a sequence of classifiers that qualifies a term, a variable, or an evaluation context. The sequence corresponds to the nesting of code expressions and the ordering of stages. Our language λ_{1v}^α restricts levels to at most one classifier:

- The empty level, written 0, corresponds to the present stage, in which the top-level program expression is run.
- A level consisting of one classifier α, written α by a slight abuse of notation, corresponds to a future stage, in which code values of the form $\langle \ldots \rangle^\alpha$ are run (Taha and Nielsen 2003).

We have no nested future stages (see Definition 3.2 below), so we can express code generators but no generators of code generators. Even with this restriction, it may still be useful to keep track of several classifiers at once, to write code such as

$$(\alpha)\langle \lambda x.\, {\sim}(f(\mathsf{run}(\beta)\langle 1 \rangle^\beta)[\alpha]) \rangle^\alpha \quad (5)$$

but not

$$(\alpha)\langle \lambda x.\, {\sim}(f(\mathsf{run}(\beta)\langle x \rangle^\beta)[\alpha]) \rangle^\alpha. \quad (6)$$

Named levels can allow (5) yet prevent the attempt to run open code in (6), whereas merely numbered levels cannot.

$$\frac{}{\Gamma \vdash i^0 : \mathsf{int}} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash i^\alpha : \mathsf{int}} \qquad \frac{(x^0 : t) \in \Gamma}{\Gamma \vdash x^0 : t} \qquad \frac{(x^\alpha : u) \in \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash x^\alpha : u}$$

$$\frac{\Gamma, x^A : t_1 \vdash e^A : t_2}{\Gamma \vdash (\lambda x^A : t_1 . e^A) : t_1 \to t_2} \qquad \frac{\Gamma, f^A : t_1 \to t_2, x^A : t_1 \vdash e^A : t_2}{\Gamma \vdash (\mathsf{fix}\, f^A (x^A : t_1) : t_2 .\, e^A) : t_1 \to t_2}$$

$$\frac{\Gamma \vdash e_1^A : \mathsf{int} \quad \Gamma \vdash e_2^A : \mathsf{int}}{\Gamma \vdash e_1^A + e_2^A : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1^A : t_1 \to t_2 \quad \Gamma \vdash e_2^A : t_1}{\Gamma \vdash e_1^A e_2^A : t_2}$$

$$\frac{\Gamma, \alpha \vdash e^0 : t}{\Gamma \vdash (\alpha) e^0 : (\alpha) t} \qquad \frac{\Gamma \vdash e^0 : (\alpha) t \quad \beta \in \Gamma}{\Gamma \vdash e^0[\beta] : t\,[\alpha := \beta]} \qquad \frac{\Gamma \vdash e^0 : (\alpha) \langle u \rangle^\alpha}{\Gamma \vdash \mathsf{run}\, e^0 : (\alpha) u}$$

$$\frac{\Gamma \vdash e^\alpha : u}{\Gamma \vdash \langle e^\alpha \rangle^\alpha : \langle u \rangle^\alpha} \qquad \frac{\Gamma \vdash e^0 : \langle u \rangle^\alpha \quad \alpha \in \Gamma}{\Gamma \vdash (\sim e^0)^\alpha : u} \qquad \frac{\Gamma \vdash e^0 : u}{\Gamma, \alpha, \Gamma' \vdash (\% e^0)^\alpha : u}$$

Figure 2. Type system of λ^α_{1v}. We assume that the names of all variables and classifiers are unique.

To unclutter the notation, we often drop classifiers where they are easy to infer. For example, we will write $(\alpha)\langle\sim\langle 1\rangle^\alpha\rangle^\alpha$ as merely $(\alpha)\langle\sim\langle 1\rangle\rangle$. The syntax of λ^α_{1v} disallows escape and CSP expressions at level 0, and the only way for a level-0 expression to contain a non-level-0 expression is to contain a bracket expression, so escape and CSP can only occur inside brackets in a program.

The classification of expressions into values depends on the level of the expression. At level 0, the only values are literals, functions, classifier generalization over values, and brackets that do not escape back to level 0. For example, $\langle\sim\langle 1\rangle^\alpha\rangle^\alpha$ is not a value but $\langle 1\rangle^\alpha$ and $\langle(\lambda x.x)1\rangle^\alpha$ both are. Notably, $\langle\% e^0\rangle^\alpha$ is a value for an arbitrary expression e^0; see §3.4.

3.1 Type system

Figure 2 shows the type system of λ^α_{1v}. A type environment Γ is an *ordered* sequence of variable and classifier bindings. We sometimes omit the empty environment [], from which all environments are built. We adopt the convention that all classifier names, like variable names, are unique. Thus, any environment of the form Γ, α, Γ' requires $\alpha \notin \Gamma$, but α can be used to label bindings in Γ'.

Just to simplify our presentation, we assume that every type uses at most one level, in the following sense.

Definition 3.1 (Used levels) The set $\mathrm{Used}(t)$ of classifiers *used* in a λ^α_{1v}-type t is defined as follows.

$$\mathrm{Used}(\mathsf{int}) = \{\} \qquad \mathrm{Used}(t_1 \to t_2) = \mathrm{Used}(t_1) \cup \mathrm{Used}(t_2) \quad (7)$$

$$\mathrm{Used}(\langle u\rangle^\alpha) = \{\alpha\} \qquad \mathrm{Used}((\alpha)t) = \mathrm{Used}(t) \setminus \{\alpha\} \quad (8)$$

For example, $\mathrm{Used}(\langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{int}\rangle^\alpha)$ is $\{\alpha\}$.

Definition 3.2 (Restriction to one level) Wherever a λ^α_{1v}-type t appears (in a term, type, binding, or judgment), we restrict $\mathrm{Used}(t)$ to be either empty or a singleton $\{\alpha\}$.

To rule out nested future stages, it is not enough to syntactically exclude terms with nested brackets, because of CSP. For example, the term $\mathsf{let}\ f = \lambda x.\langle\sim x + 1\rangle\ \mathsf{in}\ \langle\lambda x.\% f x\rangle$ has no nested brackets, but it is equivalent to $\langle\lambda x.\langle\sim x + 1\rangle\rangle$, which has nested brackets. Therefore, our restriction to one level must be stated using types.

The most important typing rules for us are those for bracket $\langle e\rangle$, escape $\sim e$, and CSP $\% e$, because they are the main way to move between levels and to interact with variable and classifier bindings. The typing derivation in Fig. 3 illustrates these rules. The leftmost branch of the proof derives the *eta* example in §2.5, except changing the type int to bool so as to apply *eta* to the identity function.

The typing rule for CSP $\% e$ rejects terms such as $\langle\lambda x.\%\langle x\rangle\rangle$, which cannot be run even though it type-checks in MetaOCaml. In exchange for not generating such un-runnable code, we gain a much simpler operational semantics, which we now turn to.

3.2 Operational semantics

Figure 4 gives the small-step operational semantics of λ^α_{1v}, using the definition of contexts in Fig. 1. A context $C^A[\,]$ can be plugged with an expression e^A to give a level-0 expression. All redexes are level-0 expressions except in the second-to-last transition. Of course, evaluation can get stuck, as on the programs $1[\beta]$ and $1(2)$.

The last transition, of the redex $\mathsf{run}\,(\alpha)\langle v^\alpha\rangle^\alpha$, *demotes* v^α as defined in the second half of the figure. Demotion homomorphically maps values v^α to level-0 expressions e^0, by replacing the level α by 0 in expression annotations and converting $\% e^0$ into e^0. (We could have simplified the run transition, inherited from λ^α, to just $C^0[\mathsf{run}\,(\alpha)\langle v^\alpha\rangle^\alpha] \rightsquigarrow C^0[v^\alpha\!\downarrow]$.)

Staging complicates our operational semantics by making our redexes possibly open and our evaluation contexts possibly binding. For example, the program $(\alpha)\langle\lambda x^\alpha{:}\mathsf{int}.\sim((\lambda y.y)\langle x^\alpha\rangle^\alpha)\rangle^\alpha$ decomposes into the open redex $(\lambda y.y)\langle x^\alpha\rangle^\alpha$ and the context $(\alpha)\langle\lambda x^\alpha{:}\mathsf{int}.\sim[\,]\rangle^\alpha$. The context binds the classifier α and then the later-stage variable x^α, whose level depends on α.

3.3 Properties

Proposition 3.3 (Subject reduction) If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$ then $\Gamma \vdash e' : t$.

Proof In particular, demotion preserves types: if $\Gamma,\ \alpha \vdash v^\alpha : u$ then $\Gamma \vdash v^\alpha\!\downarrow : u$. The latter is obvious from the rules of Fig. 2. Our typing rule for CSP makes type-preservation of demotion trivial, in stark contrast to Taha and Nielsen's (2003). $\square$

Proposition 3.4 (Determinism) If $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$ then $e_1 = e_2$.

Proposition 3.5 (Progress) If $\vdash e : t$ then either e is a value or there exists e' such that $e \rightsquigarrow e'$.

3.4 Comparison with λ^α

Our λ^α_{1v} is essentially Taha and Nielsen's λ^α (2003) with the restriction to one level, except:

- Our language is call-by-value, to better match languages in actual use such as MetaOCaml.
- We add fix, to model an effect, namely nontermination. We make this addition to show that our translation preserves the stage at which effects happen, by showing that it preserves whether a program terminates. As the first example in §2 demonstrates, it is important to evaluate an escape expression and incur its effects when the enclosing bracket is built, not run.
- We give our dynamic semantics by small-step transition rules using evaluation contexts, rather than big-step reduction rules that apply anywhere in the program. This choice is because we want to add state and control effects eventually, which is easier when transitions represent evaluation contexts explicitly and do not impose the nesting of subexpressions on their evaluation.
- Our typing rule for CSP greatly simplifies the operational semantics, at the cost of excluding some un-runnable terms (§3.1).
- Since our language is call-by-value, we do not evaluate a persisted level-0 expression. That is, $\% e$ is always a value; for example, $\%(1+1)$ is a value and does not reduce to $\% 2$. Thus, if Ω is an infinite loop such as $(\mathsf{fix}\, f(x : \mathsf{int}).\, f x)0$, then $\langle\%\Omega\rangle^\alpha$ halts, as does $\mathsf{run}\,(\alpha)\langle\lambda x^\alpha{:}\mathsf{int}.\%\Omega\rangle^\alpha$, but not $\mathsf{run}\,(\alpha)\langle\%\Omega\rangle^\alpha$.

The last difference makes the calculus more orthogonal (a redex inside brackets is always in an escape) but no less expressive: to eval-

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\alpha,\, f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha},\, x^{\alpha}:\mathsf{int}\vdash f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha}}
\quad
\dfrac{\dfrac{}{\ldots,\, x^{\alpha}:\mathsf{int}\vdash x^{\alpha}:\mathsf{int}}}{\ldots,\, x^{\alpha}:\mathsf{int}\vdash\langle x^{\alpha}\rangle^{\alpha}:\langle\mathsf{int}\rangle^{\alpha}}
}{\alpha,\, f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha},\, x^{\alpha}:\mathsf{int}\vdash f\langle x^{\alpha}\rangle^{\alpha}:\langle\mathsf{int}\rangle^{\alpha}}
}{\alpha,\, f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha},\, x^{\alpha}:\mathsf{int}\vdash\sim\!(f\langle x^{\alpha}\rangle^{\alpha}):\mathsf{int}}
}{\alpha,\, f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha}\vdash\lambda x^{\alpha}\!:\!\mathsf{int}.\sim\!(f\langle x^{\alpha}\rangle^{\alpha}):\mathsf{int}\to\mathsf{int}}
}{\alpha,\, f:\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha}\vdash\langle\lambda x^{\alpha}\!:\!\mathsf{int}.\sim\!(f\langle x^{\alpha}\rangle^{\alpha})\rangle^{\alpha}:\langle\mathsf{int}\to\mathsf{int}\rangle^{\alpha}}
}{\alpha\vdash\lambda f\!:\!\langle\mathsf{int}\rangle^{\alpha}\!\to\!\langle\mathsf{int}\rangle^{\alpha}.\,\langle\lambda x^{\alpha}\!:\!\mathsf{int}.\sim\!(f\langle x^{\alpha}\rangle^{\alpha})\rangle^{\alpha}:(\langle\mathsf{int}\rangle^{\alpha}\to\langle\mathsf{int}\rangle^{\alpha})\to\langle\mathsf{int}\to\mathsf{int}\rangle^{\alpha}}
}{\vdash(\alpha)\lambda f\!:\!\langle\mathsf{int}\rangle^{\alpha}\!\to\!\langle\mathsf{int}\rangle^{\alpha}.\langle\lambda x^{\alpha}\!:\!\mathsf{int}.\sim\!(f\langle x^{\alpha}\rangle^{\alpha})\rangle^{\alpha}\!:\!(\alpha)((\langle\mathsf{int}\rangle^{\alpha}\!\to\!\langle\mathsf{int}\rangle^{\alpha})\!\to\!\langle\mathsf{int}\!\to\!\mathsf{int}\rangle^{\alpha})}
\quad
\dfrac{
\dfrac{
\dfrac{}{\mathit{eta}:\ldots,\,\beta,\, y:\langle\mathsf{int}\rangle^{\beta}\vdash y:\langle\mathsf{int}\rangle^{\beta}}
}{\mathit{eta}:\ldots,\,\beta\vdash\lambda y\!:\!\langle\mathsf{int}\rangle^{\beta}.y:(\langle\mathsf{int}\rangle^{\beta}\to\langle\mathsf{int}\rangle^{\beta})}
}{\mathit{eta}\!:\!\ldots\vdash(\beta)\lambda y\!:\!\langle\mathsf{int}\rangle^{\beta}.y\!:\!(\beta)(\langle\mathsf{int}\rangle^{\beta}\!\to\!\langle\mathsf{int}\rangle^{\beta})}
\quad \vdots
}{\vdash\mathsf{let}\ \mathit{eta}=(\alpha)\lambda f\!:\!\langle\mathsf{int}\rangle^{\alpha}\!\to\!\langle\mathsf{int}\rangle^{\alpha}.\,\langle\lambda x^{\alpha}\!:\!\mathsf{int}.\sim\!(f\langle x^{\alpha}\rangle^{\alpha})\rangle^{\alpha}\ \mathsf{in}\ \mathsf{let}\ \mathit{id}=(\beta)\lambda y\!:\!\langle\mathsf{int}\rangle^{\beta}.y\ \mathsf{in}\ (\alpha)(\mathit{eta}[\alpha](\mathit{id}[\alpha])):(\alpha)\langle\mathsf{int}\to\mathsf{int}\rangle^{\alpha}}
$$

Figure 3. An example typing derivation in λ^{α}_{1v}

$$
\begin{aligned}
C^0[i_1+i_2] &\rightsquigarrow C^0[i_1\dot{+}i_2]\\
C^0[(\lambda x\!:\!t.\,e)v] &\rightsquigarrow C^0[e\,[x:=v]]\\
C^0[(\mathsf{fix}\,f(x:t_1):t_2.\,e)v] &\rightsquigarrow C^0[e\,[f:=\mathsf{fix}\,f(x:t_1):t_2.\,e]\,[x:=v]]\\
C^0[((\alpha)e)[\beta]] &\rightsquigarrow C^0[e\,[\alpha:=\beta]]\\
C^{\alpha}[\sim\langle v^{\alpha}\rangle^{\alpha}] &\rightsquigarrow C^{\alpha}[v^{\alpha}]\\
C^0[\mathsf{run}\,(\alpha)\langle v^{\alpha}\rangle^{\alpha}] &\rightsquigarrow C^0[(\alpha)(v^{\alpha}\!\downarrow)]
\end{aligned}
$$

$$
\begin{aligned}
i^{\alpha}\!\downarrow &= i^0 & (e_1^{\alpha}+e_2^{\alpha})\!\downarrow &= e_1^{\alpha}\!\downarrow+e_2^{\alpha}\!\downarrow\\
x^A\!\downarrow &= x^A & (e_1^{\alpha}e_2^{\alpha})\!\downarrow &= e_1^{\alpha}\!\downarrow e_2^{\alpha}\!\downarrow\\
(\%e^0)^{\alpha}\!\downarrow &= e^0 & (\lambda x^{\alpha}\!:\!u.\,e^{\alpha})\!\downarrow &= \lambda x^0\!:\!u.\,(e^{\alpha}\!\downarrow[x^{\alpha}:=x^0])
\end{aligned}
$$

$$
\begin{aligned}
&(\mathsf{fix}\,f^{\alpha}(x^{\alpha}:u_1):u_2.\,e^{\alpha})\!\downarrow\\
&\qquad= \mathsf{fix}\,f^0(x^0:u_1):u_2.\,(e^{\alpha}\!\downarrow[f^{\alpha}:=f^0]\,[x^{\alpha}:=x^0])
\end{aligned}
$$

Figure 4. Operational semantics: small-step transitions $e\rightsquigarrow e'$ and demotions $v^{\alpha}\!\downarrow$. The latter is an inductively defined map from values v^{α} to expressions e^0.

Variables	x,y,z,r,f,q
Type variables	π
Types	$\tau ::= \pi \mid \mathsf{int} \mid \tau_1\to\tau_2 \mid \forall\pi.\,\tau \mid (\vec{\tau})$
Environments	$\gamma ::= [] \mid \gamma,\pi \mid \gamma, x:\tau$
Named environments	$\rho ::= [] \mid \rho,\pi \mid \rho, x^A:\tau$
Expressions	$\varepsilon ::= i \mid x \mid \lambda x\!:\!\tau.\,\varepsilon \mid \mathsf{fix}\,f[\vec{\pi}](x:\tau_1):\tau_2.\,\varepsilon$
	$\mid \varepsilon+\varepsilon \mid \varepsilon\varepsilon \mid \Lambda\pi.\,\varepsilon \mid \varepsilon[\tau] \mid (\vec{\varepsilon}) \mid \varepsilon.i$

Figure 5. Syntax of F_2

uate the persisted term, we can write $\%e$ as $\sim(\mathsf{let}\ x=e\ \mathsf{in}\ \langle\%x\rangle)$. Conversely, if we had made CSP evaluate the persisted term, then to avoid that evaluation, we could write $\%e$ as $(\%(\lambda x\!:\!\mathsf{int}.\,e))0$.

Calcagno et al. (2004) designed a few successors to λ^{α}, with Hindley-Milner polymorphism, inference, principal types and typings. Modulo the treatment of polymorphism, those calculi are designed to simplify λ^{α} so that terms do not mention classifiers. Neither variable bindings nor brackets are labeled in those calculi; their levels are all inferred. A classifier is treated like a type variable, and an α-closed type is introduced and eliminated like a type polymorphic in α. Although this approach is attractive in the practical setting of a Hindley-Milner language such as MetaOCaml, we keep the classifiers explicit here to guide our translation. We do not concern ourselves with polymorphism and inference in this paper. We consider our calculus λ^{α}_{1v} as a desugared programming language, with all classifiers made explicit and with all polymorphism eliminated by monomorphization (inlining and type instantiation).

4. The target language F_2

Our target language is System F, also known as F_2, with products (tuples) and the fixpoint. Figure 5 reviews the syntax we use. We write $\vec{a}$ for $a_1,\ldots a_n$, the sequence of zero or more objects a_i. As particular cases of tuples, a tuple with one object is that object itself and the tuple with 0 objects is unit. We also sometimes use pattern matching as syntactic sugar for selecting parts of a tuple; for example, $\lambda(x,y)\!:\!(\mathsf{int},\mathsf{bool}).\,(y,x)$ is short for $\lambda z\!:\!(\mathsf{int},\mathsf{bool}).\,(z.2,z.1)$.

We omit our static and dynamic semantics for F_2, which are standard and call-by-value. (It does not matter whether the dynamic semantics evaluates under a type abstraction, because we only perform type abstraction immediately around a term abstraction and always apply the result of a type application to a term argument.)

At the term level, this target language adds polymorphism (including polymorphic recursion) and tuples to λ^{α}_{1v} and removes staging and classifiers. A type environment γ is an *ordered* sequence of type- and term-variable bindings.

The output of our translation uses tuples just to represent later-stage environments, for which we introduce some syntactic sugar.

Definition 4.1 (Environments as tuples) Let $\gamma=\vec{x}:\vec{\tau}$ be an F_2 environment that binds no type variable. We write the *environment abstraction* term $\lambda\gamma.\,\varepsilon$ to mean $\lambda(\vec{x})\!:\!(\vec{\tau}).\,\varepsilon$, where γ binds into ε. We also write γ for the tuple term $\vec{x}$ and the tuple type $\vec{\tau}$. Thus, $\gamma\to\tau'$ means the type $(\vec{\tau})\to\tau'$, and if ε has the type $\gamma\to\tau'$ then $\varepsilon\gamma$ means the application term $\varepsilon(\vec{x})$.

For example, if γ is $r:\pi,\ f:\pi\to\mathsf{int}$, then the abstraction $\lambda\gamma.\,fr$ is the term $\lambda(r,f)\!:\!(\pi,\pi\to\mathsf{int}).\,fr$, of type $(\pi,\pi\to\mathsf{int})\to\mathsf{int}$. As a special case, if γ is empty, then $\lambda\gamma.\,\varepsilon$ is $\lambda y\!:\!().\,\varepsilon$ and $\varepsilon\gamma$ is $\varepsilon()$.

Our translation uses classifiers in an auxiliary data structure, *named environments* ρ. As defined in Fig. 5, a named environment decorates every term-variable binding with a level. Of course, classifiers do not appear in the output of the translation.

Definition 4.2 (Restriction) Suppose ρ is a named environment and A is a named level. The *full restriction* $\rho\,\|\,A$ of ρ to A is the F_2-environment consisting of the type variables in ρ and the term variables in ρ that are decorated with A. The *incremental restriction*

$\rho|A$ of ρ to A is the F_2-environment consisting only of the term variables in ρ that are decorated with A.

Definition 4.3 (Extension) An environment, named environment, or named level *extends* another if the latter is a prefix of the former.

5. The formal translation

We define our translation from $\lambda^{\alpha}_{1v}$to F_2 by induction: first on types and environments, then on typing derivations.

5.1 Translating types and environments

Definition 5.1 (Extension by used classifiers) Given a λ^{α}_{1v}-type t and a named environment ρ, the result of *extending ρ by the classifiers used in t* is the named environment

$$\rho, \vec{\pi}(t) = \begin{cases} \rho,\, \pi,\, r^{\alpha} : \pi & \text{if } \mathrm{Used}(t) = \{\alpha\} \\ \rho & \text{if } \mathrm{Used}(t) = \{\} \end{cases} \tag{9}$$

where the type variable π and the term variable r are fresh in F_2. We write $\vec{\pi}$ for a sequence of *zero or one* type variables; a longer sequence is ruled out by the one-level restriction (Definition 3.2).

Definition 5.2 (Translating types) In a named environment ρ, a λ^{α}_{1v}-type t translates to an F_2-type $\rho\,;t$ as follows.

$$\rho\,; \mathsf{int} = \mathsf{int} \tag{10}$$
$$\rho\,; (t_1 \to t_2) = \forall\vec{\pi}.\, (\rho'\,;t_1) \to (\rho'\,;t_2) \quad \text{where } \rho' = \rho, \vec{\pi}(t_1) \tag{11}$$
$$\rho\,; (\langle u\rangle^{\alpha}) = (\rho|\alpha) \to u \tag{12}$$
$$\rho\,; ((\alpha)t) = \rho\,;t \qquad \text{where } \alpha \text{ is fresh} \tag{13}$$

In particular, if t_1 uses no classifier, then $\rho\,;(t_1 \to t_2) = (\rho\,;t_1) \to (\rho\,;t_2)$ homomorphically. For a flat type u, we have $\rho\,;u = u$.

The $\forall\vec{\pi}$ in (11) is reminiscent of Gödel's translation from intuitionistic logic to S4 (1933). For example, in the empty named environment, the λ^{α}_{1v}-type

$$(\alpha)(\langle\mathsf{int}\rangle^{\alpha} \to \langle\mathsf{bool}\rangle^{\alpha}) \to \langle\mathsf{int} \to \mathsf{bool}\rangle^{\alpha} \tag{14}$$

translates to the F_2-type (cf. §2.5)

$$\forall\pi.\, (\forall\pi'.\, ((\pi,\pi') \to \mathsf{int}) \to ((\pi,\pi') \to \mathsf{bool})) \to (\pi \to \mathsf{int} \to \mathsf{bool}) \tag{15}$$

because $\mathrm{Used}(\langle\mathsf{int}\rangle^{\alpha} \to \langle\mathsf{bool}\rangle^{\alpha}) = \mathrm{Used}(\langle\mathsf{int}\rangle^{\alpha}) = \{\alpha\}$.

Given the type translation in (11) and (13), the environment translation below is not surprising, as in the deduction theorem.

Definition 5.3 (Translating environments) A λ^{α}_{1v}-environment Γ translates to a named environment $\lfloor\Gamma\rfloor$ as follows.

$$\lfloor[]\rfloor = [] \qquad \lfloor\Gamma, x\!:\!t\rfloor = \lfloor\Gamma\rfloor, \vec{\pi}(t), x\!:\!(\lfloor\Gamma\rfloor, \vec{\pi}(t)\,;t) \tag{16}$$
$$\lfloor\Gamma, \alpha\rfloor = \lfloor\Gamma\rfloor \quad \lfloor\Gamma, x^{\alpha}\!:\!u\rfloor = \lfloor\Gamma\rfloor, x^{\alpha}\!:\!u \tag{17}$$

Thus $\lfloor\Gamma_2\rfloor$ extends $\lfloor\Gamma_1\rfloor$ whenever Γ_2 extends Γ_1.

The rest of this section defines the *coercions* motivated with escapes in §2.4. The coercion function $\rho_1 \hookrightarrow \rho_2\,;t$ is a certain injection in F_2 from $\rho_1\,;t$ to $\rho_2\,;t$. The precise definition of coercions is less important than their availability, summarized below.

Proposition 5.4 (Coercions are total and compositional) Given any λ^{α}_{1v}-type t, as long as the named environment ρ_2 extends ρ_1, the coercion $\rho_1 \hookrightarrow \rho_2\,;t$ is a total function from $\rho_1\,;t$ to $\rho_2\,;t$. If ρ_3 further extends ρ_2, then the coercion $\rho_1 \hookrightarrow \rho_3\,;t$ is equivalent to the composition $(\rho_2 \hookrightarrow \rho_3\,;t) \circ (\rho_1 \hookrightarrow \rho_2\,;t)$.

Put differently, coercions constructively show that extension among named environments induces injection among translated types.

First we show how isomorphism among named environments induces isomorphism among translated types. If $\rho_1|\alpha = \rho_2|\alpha$ for every classifier α used by t, then $\rho_1\,;t = \rho_2\,;t$. More generally, if we can convert between $\rho_1|\alpha$ and $\rho_2|\alpha$ for every classifier α used by t, then we can convert between $\rho_1\,;t$ and $\rho_2\,;t$. If t uses no classifier, then it is trivial to convert between $\rho_1\,;t$ and $\rho_2\,;t$ because they are equal. If t uses a classifier, then the conversion is barely nontrivial.

Definition 5.5 (Conversion) Let t be a λ^{α}_{1v}-type and α be a level. Let ρ_1 and ρ_2 be two named environments with the same type-variable bindings, such that $\rho_1|\beta = \rho_2|\beta$ for every level β used by t except possibly α. Suppose ε and $\bar{\varepsilon}$ are two F_2-terms such that

$$\rho_1 \parallel \alpha \vdash \varepsilon : (\rho_2|\alpha), \qquad \rho_2 \parallel \alpha \vdash \bar{\varepsilon} : (\rho_1|\alpha). \tag{18}$$

In words, ε has the tuple type $(\rho_2|\alpha)$ in the environment $\rho_1 \parallel \alpha$, and $\bar{\varepsilon}$ has the tuple type $(\rho_1|\alpha)$ in the environment $\rho_2 \parallel \alpha$. Then we define a pair of *conversion* functions in F_2

$$\mu t : (\rho_1\,;t) \to (\rho_2\,;t), \qquad \bar{\mu} t : (\rho_2\,;t) \to (\rho_1\,;t) \tag{19}$$

simultaneously by the following induction on the structure of t. We omit the definition of $\bar{\mu}t$ by symmetry with μt.

$$\mu\,\mathsf{int} = \lambda x{:}\mathsf{int}.\, x \tag{20}$$
$$\mu(t_1 \to t_2) = \lambda f.\, \Lambda\vec{\pi}.\, \lambda x.\, \mu t_2(f[\vec{\pi}](\bar{\mu} t_1(x))) \quad \text{where the length of } \vec{\pi} \text{ is } \#\mathrm{Used}(t_1) \tag{21}$$
$$\mu(\langle u\rangle^{\alpha}) = \lambda f.\, \lambda(\rho_2|\alpha).\, f(\bar{\varepsilon}) \tag{22}$$
$$\mu((\alpha)t) = \mu t \qquad \text{where } \alpha \text{ is fresh} \tag{23}$$

We notate the conversion μ as $\mathrm{Conv}(\rho_1, \rho_2, \alpha, \varepsilon, \bar{\varepsilon})$.

For example, if

$$\rho_1 = \pi, x^{\alpha} : \pi, y^{\alpha} : \mathsf{int}, \quad \rho_2 = \pi, f : \mathsf{int} \to \pi, z^{\alpha} : (\mathsf{int}, \pi) \tag{24}$$

and we let μ be $\mathrm{Conv}(\rho_1, \rho_2, \alpha, (y,x), (z.2, z.1))$, then

$$\mu(\langle\mathsf{bool}\rangle^{\alpha}) = \lambda f{:}(\pi,\mathsf{int}) \to \mathsf{bool}.\, \lambda z{:}(\mathsf{int},\pi).\, f(z.2, z.1), \tag{25}$$
$$\bar{\mu}(\langle\mathsf{bool}\rangle^{\alpha}) = \lambda f{:}(\mathsf{int},\pi) \to \mathsf{bool}.\, \lambda(x,y){:}(\pi,\mathsf{int}).\, f(y,x). \tag{26}$$

Definition 5.6 (Coercion) Suppose ρ_1 and ρ_2 are named environments and ρ_2 extends ρ_1. For every λ^{α}_{1v}-type t, we define a *coercion* function in F_2 from $\rho_1\,;t$ to $\rho_2\,;t$, that is, a term of type $\rho_1\,;t \to \rho_2\,;t$ in the environment $\rho_2 \parallel A$. We notate this coercion as $\rho_1 \hookrightarrow \rho_2\,;t$.

By induction on the difference between ρ_1 and ρ_2, we consider two cases then compose the coercions. First, if ρ_2 is ρ_1 or ρ_1, π or $\rho_1, y : \tau$, then the coercion is the identity. Second, if $\rho_2 = \rho_1, y^{\beta} : \tau$, then we define the coercion by the following induction on t.

$$\rho_1 \hookrightarrow \rho_2\,; \mathsf{int} = \lambda x{:}\mathsf{int}.\, x \tag{27}$$
$$\rho_1 \hookrightarrow \rho_2\,; (t_1 \to t_2) = \text{see below} \tag{28}$$
$$\rho_1 \hookrightarrow \rho_2\,; (\langle u\rangle^{\alpha}) = \lambda f.\, \lambda(\rho_2|\alpha).\, f(\rho_1|\alpha) \tag{29}$$
$$\rho_1 \hookrightarrow \rho_2\,; ((\alpha)t) = \rho_1 \hookrightarrow \rho_2\,;t \quad \text{where } \alpha \text{ is fresh} \tag{30}$$

In (28), we define the coercion $\rho_1 \hookrightarrow \rho_2\,;(t_1 \to t_2)$ by considering two cases. On one hand, if β is not used in t_1 and so $(\rho_1, \vec{\pi}(t_1))\,;t_1 = (\rho_2, \vec{\pi}(t_1))\,;t_1$, then the coercion is

$$\lambda f.\, \Lambda\vec{\pi}.\, \lambda x.\, (\rho_1, \vec{\pi}(t_1) \hookrightarrow \rho_1, \vec{\pi}(t_1), y^{\beta} : \tau\,;t_2)(f[\vec{\pi}](x)). \tag{31}$$

On the other hand, if $\mathrm{Used}(t_1) = \{\beta\}$, then let

$$\rho_1' = \rho_1, \vec{\pi}(t_1) = \rho_1,\, \pi,\, r^{\beta} : \pi, \tag{32}$$
$$\rho_2' = \rho_2, \vec{\pi}(t_1) = \rho_1,\, y^{\beta} : \tau,\, \pi,\, r^{\beta} : \pi, \tag{33}$$

following Definition 5.1. Then by Definition 5.2,

$$\rho_1\,; (t_1 \to t_2) = \forall\pi.\, (\rho_1'\,;t_1 \to \rho_1'\,;t_2), \tag{34}$$
$$\rho_2\,; (t_1 \to t_2) = \forall\pi.\, (\rho_2'\,;t_1 \to \rho_2'\,;t_2). \tag{35}$$

Define the named environment $\rho_1'' = \rho_1,\, \pi,\, r'^{\beta} : (\tau, \pi)$ and let μ be the conversion $\mathrm{Conv}(\rho_1'', \rho_2', \beta, (\rho_1|\beta, r'.1, r'.2), (\rho_1|\beta, (y, r)))$. We then define the coercion from the type (34) to the type (35) to be

$$\lambda f.\,\Lambda\pi.\ \lambda x.\,\mu t_2(f[(\tau,\pi)](\bar\mu t_1(x))). \tag{36}$$

Examples A simple example of a coercion is $\lfloor\alpha\rfloor \hookrightarrow \lfloor\alpha, x^\alpha : \mathsf{int}\rfloor\,;\ \langle\mathsf{bool}\rangle^\alpha$, where $\lfloor\alpha\rfloor$ is $[]$ and $\lfloor\alpha,\ x^\alpha : \mathsf{int}\rfloor$ is $x^\alpha : \mathsf{int}$ by Definition 5.3. The coercion is an F_2-function from the type $[]\,;\langle\mathsf{bool}\rangle^\alpha = () \to \mathsf{bool}$ to the type $x^\alpha : \mathsf{int}\,;\langle\mathsf{bool}\rangle^\alpha = \mathsf{int} \to \mathsf{bool}$, namely $\lambda f.\,\lambda x.\,f()$ by (29) and (the bool analogue of) (27).

A more involved example of a coercion is

$$(\lfloor\alpha\rfloor,\,\pi,\,r^\alpha : \pi) \hookrightarrow \lfloor\alpha,\,f : \langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{bool}\rangle^\alpha,\,x^\alpha : \mathsf{int}\rfloor;$$
$$(\langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{bool}\rangle^\alpha). \tag{37}$$

This coercion occurs in the translation of the *eta* function in §2.5. Here $\lfloor\alpha\rfloor$ is $[]$ and $\lfloor\alpha,\,f : \langle\mathsf{int}\rangle^\alpha \to \langle\mathsf{bool}\rangle^\alpha,\,x^\alpha : \mathsf{int}\rfloor$ is

$$\pi,\,r^\alpha{:}\,\pi,\,f : \forall\pi'.\,((\pi,\pi') \to \mathsf{int}) \to ((\pi,\pi') \to \mathsf{bool}),\,x^\alpha{:}\,\mathsf{int} \tag{38}$$

by Definition 5.3. The coercion (37) is a function in F_2 from $\forall\pi'.\,((\pi,\pi') \to \mathsf{int}) \to ((\pi,\pi') \to \mathsf{bool})$ to $\forall\pi'.\,((\pi,\mathsf{int},\pi') \to \mathsf{int}) \to ((\pi,\mathsf{int},\pi') \to \mathsf{bool})$. It is equivalent to the F_2-term

$$\lambda f.\,\Lambda\pi'.\ \lambda f_1.\,\mathsf{let}\ f_2 = f[(\mathsf{int},\pi')](\lambda(x,(y,z)).\,f_1(x,y,z))\ \mathsf{in}$$
$$\lambda(x,y,z).\,f_2(x,(y,z)).$$

5.2 Translating terms and derivations

Our translation of terms and derivations is guided by our translation of types and environments, so this section should be read in conjunction with the previous one. We want to translate a λ_{1v}^{α}-judgment $\Gamma \vdash e^0 : t$ to an F_2-judgment $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon : (\lfloor\Gamma\rfloor\,;t)$. However, to handle code with escaping and cross-stage persistence inductively, we also translate a λ_{1v}^{α}-judgment $\Gamma \vdash e^\alpha : u$ to a judgment $\lfloor\Gamma\rfloor \parallel \alpha,\,q_1 : \tau_1,\ldots,q_n : \tau_n \vdash \varepsilon : u$ along with an ordered sequence of n *auxiliary* judgments $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon_i : \tau_i$. As detailed in the following definition and illustrated in the following example, the translation of brackets combines—that is, flattens—these judgments into let.

Definition 5.7 (Translating terms, judgments, and derivations) The translation of the source derivation is defined inductively over the source derivation, or, equivalently, on the source term e^A. We mark some applications $\varepsilon_1\varepsilon_2$ as *administrative* by writing them as $\varepsilon_1 \mathbin{@} \varepsilon_2$. All elimination forms in conversions and coercions are also administrative. See also the translation of bracket.

Literal constant: Translate the judgment $\Gamma \vdash i^A : \mathsf{int}$ to the judgment $\lfloor\Gamma\rfloor \parallel A \vdash i : \mathsf{int}$, with no auxiliary judgment if A is not empty.

Variable: Translate $\Gamma \vdash x^\alpha : u$ to $\lfloor\Gamma\rfloor \parallel \alpha \vdash x : u$, with no auxiliary judgment. Translate the judgment $\Gamma \vdash x^0 : t$, where $\Gamma = \Gamma_1, x^0 : t, \Gamma_2$, to the judgment $\lfloor\Gamma\rfloor \parallel 0 \vdash (\lfloor\Gamma_1\rfloor, \vec\pi(t) \hookrightarrow \lfloor\Gamma\rfloor\,;t) \mathbin{@} x : (\lfloor\Gamma\rfloor\,;t)$.

Abstraction: Let Γ be $\Gamma_1,\,x^A : t_1$ and t be $t_1 \to t_2$. Suppose the judgment $\Gamma \vdash e^A : t_2$ translates to $\lfloor\Gamma\rfloor \parallel A,\,\vec q : \vec\tau \vdash \varepsilon : \tau$, with some auxiliary judgments if A is not empty. If A is not empty, then $\tau = t_2$ and $\lfloor\Gamma_1\rfloor \parallel 0 = \lfloor\Gamma\rfloor \parallel 0$, so just translate $\Gamma_1 \vdash (\lambda x^A{:}t_1.e^A) : t$ to $\lfloor\Gamma_1\rfloor \parallel A,\,\vec q : \vec\tau \vdash (\lambda x{:}t_1.\,\varepsilon) : t$, with the same auxiliary judgments. If A is empty, then $\tau = \lfloor\Gamma_1\rfloor, \vec\pi(t_1)\,; t_2$, so translate $\Gamma_1 \vdash (\lambda x{:}t_1.\,e) : t$ to $\lfloor\Gamma_1\rfloor \parallel 0 \vdash (\Lambda\vec\pi(t_1).\ \lambda x{:}(\lfloor\Gamma_1\rfloor,\vec\pi(t_1)\,;t_1).\,\varepsilon) : (\lfloor\Gamma_1\rfloor\,;t)$.

Fixpoint: Let Γ be $\Gamma_1,\,f^A : t,\,x^A : t_1$ where t is $t_1 \to t_2$. Suppose $\Gamma \vdash e^A : t_2$ translates to $\lfloor\Gamma\rfloor \parallel A,\,\vec q : \vec\tau \vdash \varepsilon : \tau$, with some auxiliary judgments if A is not empty. If A is not empty, then $\tau = t_2$ and $\lfloor\Gamma_1\rfloor \parallel 0 = \lfloor\Gamma\rfloor \parallel 0$, so just translate $\Gamma_1 \vdash (\mathsf{fix} f^A(x^A : t_1) : t_2.\ e^A) : t$ to $\lfloor\Gamma_1\rfloor \parallel A,\,\vec q : \vec\tau \vdash (\mathsf{fix} f(x : t_1) : t_2.\ \varepsilon) : t$, with the same auxiliary judgments. If A is empty, then $\Gamma_1,\,f : t,\,x : t_1 \vdash e : t_2$ translates to

$$\lfloor\Gamma_1\rfloor \parallel 0,\,\vec\pi,\,f : (\lfloor\Gamma_1\rfloor,\vec\pi(t)\,;t),\,\vec\pi',\,x : (\rho\,;t_1) \vdash \varepsilon : (\rho\,;t_2), \tag{39}$$

where $\rho = \lfloor\Gamma_1\rfloor, \vec\pi(t), \vec\pi'(t_1)$. The one-level restriction (Definition 3.2) leaves three possibilities for $\mathrm{Used}(t_1)$ and $\mathrm{Used}(t_2)$. First, if $\mathrm{Used}(t) = \mathrm{Used}(t_1) = \mathrm{Used}(t_2) = \{\}$, then just translate $\Gamma_1 \vdash (\mathsf{fix} f(x).\ e) : t$ to $\lfloor\Gamma_1\rfloor \parallel 0 \vdash \mathsf{fix} f(x).\ \varepsilon : (\lfloor\Gamma_1\rfloor\,;t)$. Second, if $\mathrm{Used}(t) = \mathrm{Used}(t_1) = \{\alpha\} \supseteq \mathrm{Used}(t_2)$, then $\rho = \lfloor\Gamma_1\rfloor,\,\pi,\,r^\alpha{:}\,\pi,\,\pi',\,r'^\alpha : \pi'$. Define the named environment $\rho_1 = \lfloor\Gamma_1\rfloor,\,\pi,\,r^\alpha : \pi$ and let μ be the conversion $\mathrm{Conv}(\rho_1, (\rho_1,\ r'^\alpha : ()), \alpha, (\rho_1|\alpha, ()), (\rho_1|\alpha))$. Then translate $\Gamma_1 \vdash (\mathsf{fix} f(x).\ e) : t$ to

$$\lfloor\Gamma_1\rfloor \parallel 0 \vdash (\mathsf{fix} f[\pi](x' : \rho_1\,;t_1) : (\rho_1\,;t_2).$$
$$\bar\mu t_2 \mathbin{@} (\varepsilon\,[\pi' := ()]\,[x := \mu t_1 \mathbin{@} x'])) : (\lfloor\Gamma_1\rfloor\,;t). \tag{40}$$

Third, if $\mathrm{Used}(t) = \mathrm{Used}(t_2) = \{\alpha\}$ but $\mathrm{Used}(t_1) = \{\}$, then $\rho = \lfloor\Gamma_1\rfloor,\,\pi,\,r^\alpha{:}\,\pi$. Define the named environment $\rho' = \lfloor\Gamma_1\rfloor,\,r^\alpha : ()$ and let μ be the conversion $\mathrm{Conv}(\lfloor\Gamma_1\rfloor, \rho', \alpha, (\lfloor\Gamma_1\rfloor|\alpha, ()), (\lfloor\Gamma_1\rfloor|\alpha))$. Then translate $\Gamma_1 \vdash (\mathsf{fix} f(x).\ e) : t$ to

$$\lfloor\Gamma_1\rfloor \parallel 0 \vdash (\mathsf{fix} f'(x' : \lfloor\Gamma_1\rfloor\,;t_1) : (\lfloor\Gamma_1\rfloor\,;t_2).$$
$$\bar\mu t_2 \mathbin{@} (\varepsilon\,[\pi := ()]\,[f := \mu t \mathbin{@} f']\,[x := \mu t_1 \mathbin{@} x'])) : (\lfloor\Gamma_1\rfloor\,;t). \tag{41}$$

Addition: If $e_1^A : \mathsf{int}$ translates to $\varepsilon_1 : \mathsf{int}$ and $e_2^A : \mathsf{int}$ translates to $\varepsilon_2 : \mathsf{int}$ (both with some auxiliary judgments if A is not empty), then translate $e_1^A + e_2^A : \mathsf{int}$ to $\varepsilon_1 + \varepsilon_2 : \mathsf{int}$ (concatenating the two sequences of auxiliary judgments if A is not empty).

Application: Suppose that $\Gamma \vdash e_1^A : t_1 \to t_2$ translates to $\lfloor\Gamma\rfloor \parallel A$, $\vec q_1 : \vec\tau_1 \vdash \varepsilon_1 : \tau_1$ and $\Gamma \vdash e_2^A : t_1$ translates to $\lfloor\Gamma\rfloor \parallel A,\,\vec q_2 : \vec\tau_2 \vdash \varepsilon_2 : \tau_2$, both with some auxiliary judgments if A is not empty. If A is not empty, then just translate e_1e_2 to $\varepsilon_1\varepsilon_2$, concatenating the two sequences of auxiliary judgments. If A is empty, then $\mathrm{Used}(t_1)$ is either $\{\}$ or $\{\alpha\}$ by the one-level restriction (Definition 3.2). If $\mathrm{Used}(t_1)$ is empty, then just translate e_1e_2 to $\varepsilon_1\varepsilon_2$. If $\mathrm{Used}(t_1)$ is $\{\alpha\}$, then define the named environment $\rho = \lfloor\Gamma\rfloor,\,r^\alpha : ()$ and let μ be the conversion $\mathrm{Conv}(\lfloor\Gamma\rfloor, \rho, \alpha, (\lfloor\Gamma\rfloor|\alpha, ()), (\lfloor\Gamma\rfloor|\alpha))$. Translate e_1e_2 to $\mathsf{let}\ (f,x) = (\varepsilon_1, \varepsilon_2)\ \mathsf{in}\ (\bar\mu(t_1 \to t_2) \mathbin{@} (f[()]))x$.

Classifier introduction: If e translates to ε, then $(\alpha)e$ also translates to ε, where α is fresh.

Classifier instantiation: Suppose that $\Gamma \vdash e : (\alpha)t$ translates to $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon : (\lfloor\Gamma\rfloor\,;t)$, where α is fresh, and β is a classifier in Γ. Let $x_1{}^\beta : \tau_1,\ldots,x_m{}^\beta : \tau_m$ be all the term-variable bindings in $\lfloor\Gamma\rfloor$ that are decorated with β. Define the named environment $\rho = \lfloor\Gamma\rfloor$, $x_1{}^\alpha : \tau_1,\ldots,x_m{}^\alpha : \tau_m$. Then $\rho|\alpha = \lfloor\Gamma\rfloor|\beta$, so $\rho\,;t = \lfloor\Gamma\rfloor\,;(t\,[\alpha := \beta])$. Translate $e[\beta]$ to $(\lfloor\Gamma\rfloor \hookrightarrow \rho\,;t) \mathbin{@} \varepsilon$.

Run: If $\Gamma \vdash e : (\alpha)\langle u\rangle^\alpha$ (where α is fresh) translates to $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon : () \to u$, then translate $\Gamma \vdash \mathsf{run}\,e : (\alpha)u$ to $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon() : u$.

Bracket: If $\Gamma \vdash e^\alpha : u$ translates to $\lfloor\Gamma\rfloor \parallel \alpha,\,q_1 : \tau_1,\ldots,q_n : \tau_n \vdash \varepsilon : u$ with n auxiliary judgments $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon_i : \tau_i$, then translate the quotation $\Gamma \vdash \langle e^\alpha\rangle^\alpha : \langle u\rangle^\alpha$ to

$$\lfloor\Gamma\rfloor \parallel 0 \vdash \mathsf{let}\ q_1 = \varepsilon_1\ \mathsf{in}\ \ldots \mathsf{let}\ q_n = \varepsilon_n\ \mathsf{in}\ \lambda(\lfloor\Gamma\rfloor|\alpha).\,\varepsilon : (\lfloor\Gamma\rfloor|\alpha) \to u. \tag{42}$$

Mark as administrative: (a) $\mathsf{let}\ q_i = \varepsilon_i$ for auxiliary judgments from translating CSP; (b) environment tuple projections $(\lfloor\Gamma\rfloor|\alpha).i$.

Escape: If $\Gamma \vdash e : \langle u\rangle^\alpha$ translates to $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon : \lfloor\Gamma\rfloor|\alpha \to u$, then make the latter judgment the sole auxiliary judgment in translating $\Gamma \vdash {\sim}e : u$ to $\lfloor\Gamma\rfloor \parallel \alpha,\,q : (\lfloor\Gamma\rfloor|\alpha \to u) \vdash q \mathbin{@} (\lfloor\Gamma\rfloor|\alpha) : u$.

Cross-stage persistence: If $\Gamma \vdash e : u$ translates to $\lfloor\Gamma\rfloor \parallel 0 \vdash \varepsilon : u$, then make $\lfloor\Gamma,\alpha,\Gamma'\rfloor \parallel 0 \vdash \lambda().\,\varepsilon : () \to u$ the sole auxiliary judgment in translating $\Gamma,\alpha,\Gamma' \vdash \%e : u$ to $\lfloor\Gamma,\alpha,\Gamma'\rfloor \parallel \alpha,\,q : () \to u \vdash q \mathbin{@} () : u$.

Examples We return to the ef_2 example in §2.3 and §2.4. The source term is the application of the function

$$\lambda ef^0 : (\beta)(\langle\mathsf{int}\rangle^\beta \to \langle\mathsf{int} \to \mathsf{int}\rangle^\beta).$$
$$(\alpha)\langle\lambda x^\alpha.\,\lambda y^\alpha.\,{\sim}(ef[\alpha]\langle x^\alpha \times y^\alpha\rangle^\alpha)\rangle^\alpha \tag{43}$$

to the argument

$$(\alpha)\lambda z{:}\,\langle\mathsf{int}\rangle^\alpha.\,\langle\lambda x^\alpha{:}\,\mathsf{int}.\,{\sim}z + x^\alpha\rangle^\alpha. \tag{44}$$

Because the type of the argument (44) uses no level, this application translates to the application of the translations of (43) and (44) in the empty environment. We can thus translate (43) and (44) separately (thanks to *compositionality in the large*, Proposition 5.14).

Translating ef We translate the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha,\, x^\alpha : \mathsf{int} \vdash z^0 : \langle \mathsf{int}\rangle^\alpha}\,1}{\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha,\, x^\alpha : \mathsf{int} \vdash {\sim}z : \mathsf{int}}\,2 \qquad \dfrac{}{\ldots,\, x^\alpha : \mathsf{int} \vdash x^\alpha : \mathsf{int}}\,3}{\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha,\, x^\alpha : \mathsf{int} \vdash {\sim}z + x^\alpha : \mathsf{int}}\,4}{\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha \vdash \lambda x^\alpha{:}\mathsf{int}.\, {\sim}z + x^\alpha : \mathsf{int} \to \mathsf{int}}\,5}{\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha \vdash \langle \lambda x^\alpha{:}\mathsf{int}.\, {\sim}z + x^\alpha\rangle^\alpha : \langle \mathsf{int} \to \mathsf{int}\rangle^\alpha}\,6}{\alpha \vdash \lambda z^0{:}\langle \mathsf{int}\rangle^\alpha.\, \langle \lambda x^\alpha{:}\mathsf{int}.\, {\sim}z + x^\alpha\rangle^\alpha : \langle \mathsf{int}\rangle^\alpha \to \langle \mathsf{int} \to \mathsf{int}\rangle^\alpha}\,7}{\vdash (\alpha)\lambda z^0{:}\langle \mathsf{int}\rangle^\alpha.\langle \lambda x^\alpha{:}\mathsf{int}.{\sim}z{+}x^\alpha\rangle^\alpha{:}(\alpha)(\langle \mathsf{int}\rangle^\alpha{\to}\langle \mathsf{int}{\to}\mathsf{int}\rangle^\alpha)}\,8$$

from top to bottom. The top λ^α_{1v}-environment $\alpha,\, z^0 : \langle \mathsf{int}\rangle^\alpha,\, x^\alpha : \mathsf{int}$ translates to the named environment $\pi,\, r^\alpha : \pi,\, z^0 : \pi \to \mathsf{int},\, x^\alpha : \mathsf{int}$.

1. $\pi,\, z : \pi \to \mathsf{int} \vdash (\lambda f.\, \lambda (r,x) : (\pi, \mathsf{int}).\, f r) \,@\, z : (\pi, \mathsf{int}) \to \mathsf{int}$
2. $\pi,\, r : \pi,\, x : \mathsf{int},\, q : (\pi, \mathsf{int}) \to \mathsf{int} \vdash q \,@\, (r,x) : \mathsf{int}$
   with the auxiliary judgment
   $\pi,\, z : \pi \to \mathsf{int} \vdash (\lambda f.\, \lambda (r,x) : (\pi, \mathsf{int}).\, f r) \,@\, z : (\pi, \mathsf{int}) \to \mathsf{int}$
3. $\pi,\, r : \pi,\, x : \mathsf{int} \vdash x : \mathsf{int}$
4. $\pi,\, r : \pi,\, x : \mathsf{int},\, q : (\pi, \mathsf{int}) \to \mathsf{int} \vdash q \,@\, (r,x) + x : \mathsf{int}$
   with the same auxiliary judgment as in 3
5. $\pi,\, r : \pi,\, q : (\pi, \mathsf{int}) \to \mathsf{int} \vdash \lambda x{:}\mathsf{int}.\, q \,@\, (r,x) + x : \mathsf{int} \to \mathsf{int}$
   with the same auxiliary judgment as in 3
6. $\pi,\, z : \pi \to \mathsf{int} \vdash \mathsf{let}\ q = (\lambda f.\, \lambda (r,x) : (\pi, \mathsf{int}).\, f r) \,@\, z\ \mathsf{in}$
   $\lambda r.\, \lambda x{:}\mathsf{int}.\, q \,@\, (r,x) + x : \pi \to \mathsf{int} \to \mathsf{int}$
7. $\vdash \Lambda \pi.\ \lambda z{:}\pi{\to}\mathsf{int}.\, \mathsf{let}\ q = (\lambda f.\, \lambda (r,x) : (\pi, \mathsf{int}).\, f r) \,@\, z\ \mathsf{in}$
   $\lambda r.\, \lambda x{:}\mathsf{int}.\, q \,@\, (r,x) + x : \forall \pi.\, (\pi \to \mathsf{int}) \to (\pi \to \mathsf{int} \to \mathsf{int})$
8. Same as 7

The result is the same as at the end of §2.4, modulo a β-value redex.

Translating ef_2 proper In the typing derivation of (43), we focus on the application

$$\dfrac{\overset{\vdots\, 2}{\Gamma \vdash ef[\alpha] : \langle \mathsf{int}\rangle^\alpha \to \langle \mathsf{int} \to \mathsf{int}\rangle^\alpha} \qquad \overset{\vdots\, 1}{\Gamma \vdash \langle x^\alpha \times y^\alpha\rangle^\alpha : \langle \mathsf{int}\rangle^\alpha}}{\Gamma \vdash ef[\alpha]\langle x^\alpha \times y^\alpha\rangle^\alpha : \langle \mathsf{int} \to \mathsf{int}\rangle^\alpha.}\,3$$

Here the λ^α_{1v}-environment Γ is defined by

$$\Gamma = ef^0 : (\beta)(\langle \mathsf{int}\rangle^\beta \to \langle \mathsf{int} \to \mathsf{int}\rangle^\beta),\ \alpha,\ x^\alpha : \mathsf{int},\ y^\alpha : \mathsf{int} \tag{45}$$

and translates to the named environment

$$\lfloor \Gamma \rfloor = ef^0 : \forall \pi.\, (\pi \to \mathsf{int}) \to (\pi \to \mathsf{int} \to \mathsf{int}),\ x^\alpha : \mathsf{int},\ y^\alpha : \mathsf{int}. \tag{46}$$

The bracket 1 above translates easily to the judgment

$$ef : \forall \pi.\, (\pi \to \mathsf{int}) \to (\pi \to \mathsf{int} \to \mathsf{int}) \vdash \lambda (x,y).\, x \times y : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}. \tag{47}$$

To translate the classifier instantiation 2, we apply the coercion

$$\lfloor \Gamma \rfloor \hookrightarrow (\lfloor \Gamma \rfloor, x^\beta : \mathsf{int}, y^\beta : \mathsf{int})\, ;\, \langle \mathsf{int}\rangle^\beta \to \langle \mathsf{int} \to \mathsf{int}\rangle^\beta \tag{48}$$

to *ef*, to turn it from the type

$$\forall \pi.\, (\pi \to \mathsf{int}) \to (\pi \to \mathsf{int} \to \mathsf{int}) \tag{49}$$

to the type

$$\forall \pi.\, ((\mathsf{int}, \mathsf{int}, \pi) \to \mathsf{int}) \to ((\mathsf{int}, \mathsf{int}, \pi) \to \mathsf{int} \to \mathsf{int}). \tag{50}$$

The result of the coercion is essentially $\Lambda \pi.\ \lambda r.\, ef[(\mathsf{int}, \mathsf{int}, \pi)] r$. Finally, to translate the application 3, we instantiate this type variable π to $()$ and apply the conversion

$$\overline{\mathrm{Conv}(\lfloor \Gamma \rfloor, (\lfloor \Gamma \rfloor, r^\alpha : ()), \alpha, (x,y,()), (x,y))}(\langle \mathsf{int}\rangle^\alpha \to \langle \mathsf{int} \to \mathsf{int}\rangle^\alpha) \tag{51}$$

to get a function of type

$$((\mathsf{int}, \mathsf{int}) \to \mathsf{int}) \to ((\mathsf{int}, \mathsf{int}) \to \mathsf{int} \to \mathsf{int}). \tag{52}$$

We then apply this function to (47).

5.3 Properties

Proposition 5.8 (Type preservation) If $\Gamma \vdash e^0 : t$ in λ^α_{1v}, and it translates to $\lfloor \Gamma \rfloor \parallel 0 \vdash \varepsilon : (\lfloor \Gamma \rfloor\, ; t)$, then $\lfloor \Gamma \rfloor \parallel 0 \vdash \varepsilon : (\lfloor \Gamma \rfloor\, ; t)$ in F_2.

Proof Definition 5.7 amounts to a constructive proof. □

Because the type system of F_2 is sound, type preservation ensures that our translation does not go wrong. In particular, the translated term never looks up free variables, thus preventing scope extrusion.

We turn to the dynamic properties of our term translation.

Definition 5.9 An *administrative reduction* is a β-value reduction in F_2 (anywhere in a term, even under λ) of an elimination form marked as administrative in Definition 5.7.

Administrative reduction is terminating (since each step reduces the number of administrative redexes) and confluent (since there is no critical pair), so it brings every term ε to a *normal form* ANF(ε).

This normal form ANF(ε) is observationally equivalent to ε since all β-value reductions preserve observational equivalence in F_2.

Our translation preserves values in the following sense. We conjecture that it also preserves reductions and hence observations.

Proposition 5.10 (Value preservation) If $\Gamma \vdash v^0 : t$ and v translates to ε, then ANF(ε) is a value.

Conjecture 5.11 (Reduction preservation) Suppose $\Gamma \vdash e_1^0 {:} t$ and $e_1 \rightsquigarrow e_2$. If e_1 and e_2 translate to ε_1 and ε_2, then there exists ε so that ANF(ε) = ANF(ε_2) and ANF(ε_1) $\rightsquigarrow^+ \varepsilon$ in call-by-value F_2.

Corollary 5.12 (Observation preservation) If $[] \vdash e^0 {:} t$ translates to $[] \vdash \varepsilon : ([]\, ; t)$, then e terminates if and only if ε terminates.

The two conjectures above entail that our translation preserves evaluation order. Because F_2 without fix is strongly normalizing, they also entail that λ^α_{1v} without fix is strongly normalizing.

Although our translation is defined to operate on an entire λ^α_{1v} program at once, it is in fact compositional enough for parts of an expression or a modular program to be translated separately.

Proposition 5.13 (Compositionality in the small) Let e be a well-typed λ^α_{1v}-term with n subterms $e_1, \ldots, e_n$, and e' be the well-typed result of replacing $e_1, \ldots, e_n$ by $e'_1, \ldots, e'_n$. If the subterms $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$, in their environments in λ^α_{1v}, translate to the same (or equivalent) F_2-judgments, then e and e', in their environments in λ^α_{1v}, also translate to the same (or equivalent) F_2-judgments.

Proposition 5.14 (Compositionality in the large) Let Γ_1 and Γ_2 be two λ^α_{1v}-environments that differ only in their variable bindings at level 0 whose types use no classifier. If $\Gamma_1 \vdash e^0 : t$ and $\Gamma_2 \vdash e^0 : t$, then the translation of e^0 is the same in Γ_1 as in Γ_2.

Proof Observe that $\lfloor \Gamma_1 \rfloor \parallel \alpha = \lfloor \Gamma_2 \rfloor \parallel \alpha$ for all α. □

Proposition 5.13 means that the translation of a term can proceed with translating its parts in parallel. Proposition 5.14 means that top-level definitions that do not share a classifier can be translated separately. For example, the program in Fig. 3 defines the code generators *eta* and *id* at the top level. These definitions and the body that uses them can be translated separately from each other.

6. Scope extrusion

We turn in this section from our formal translation to how it helps us combine staging with effects soundly in our ongoing work.

To continue the *power* example from §2, suppose that we want to count the multiplication operations as we generate them. For example, *power7* should produce the count 5 along with the function $\lambda x. x \times square\ (x \times square\ (x \times 1))$. In λ^{α}_{1v} and in MetaOCaml without effects, this count is hard to pass out of the scope of the later-stage variable x, because the binder λx must apply to later-stage code, not an earlier-stage count.

If we add ML-style mutable references to λ^{α}_{1v}, then the counting becomes easy: we change the definition of *power* as follows.

$$\begin{aligned}&\mathsf{let}\ count = \mathrm{ref}\ 0\\ &\mathsf{let}\ power = (\alpha)\,\mathsf{fix}\, f(n:\mathsf{int}):\langle\mathsf{int}\rangle^{\alpha} \to \langle\mathsf{int}\rangle^{\alpha}.\ \lambda x:\langle\mathsf{int}\rangle^{\alpha}.\\ &\qquad\qquad \text{if } n = 0 \text{ then } \langle 1\rangle^{\alpha} \text{ else } count \leftarrow\ !count + 1;\\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{if } n \bmod 2 = 0 \ldots\end{aligned}$$

After evaluating *power7*, the number of multiplications generated can be retrieved by !*count*.

Unfortunately, adding state so naïvely to a staged language results in scope extrusion. For example, the following program generates a piece of open code *then runs it*.

$$\mathsf{run}\,(\alpha)(\mathsf{let}\ x = \mathrm{ref}\,\langle 1\rangle^{\alpha}\ \mathsf{in}\ \langle \lambda y. {\sim}(x \leftarrow \langle y\rangle^{\alpha}; \langle ()\rangle^{\alpha})\rangle^{\alpha}; !x) \qquad (53)$$

The let expression above evaluates to $\langle y\rangle^{\alpha}$ with no binder for y in sight, so the program gets stuck at $(\alpha)y$. In MetaOCaml, this example causes a type-checking error at run time.

Adding state breaks soundness because the environment where a code value is created may no longer be a prefix of the environment where it is used. In terms of our translation, we would need to coerce the type $\rho_1\,;t$ to the type $\rho_2\,;t$ when ρ_2 does not extend ρ_1. Such a coercion does not exist in general. If we translate (53) informally without such a reverse coercion, we get

$$\begin{aligned}&(\mathsf{let}\ x = \mathrm{ref}(\lambda().\,1)\ \mathsf{in}\ \mathsf{let}\ v = x \leftarrow (\lambda(y).\,y); \lambda(y).\,()\ \mathsf{in}\\ &\qquad\qquad\qquad (\lambda().\,\lambda y. v(y)); !x)(),\end{aligned}$$

which does not type-check in F_2, even without any value restriction.

These examples suggest that, to prevent scope extrusion in staged programs with effects, we can try to translate the staging away and see if we can come up with the coercions needed to keep the translation well-typed (that is, to maintain Proposition 5.8). For example, the effectful use of *count* in the *power* example above is sound, because mutation at the flat type int uses no classifiers and so needs no coercion (as discussed above Definition 5.5). Of course, other uses of effects in staging, such as for let insertion, may require more coercions or different tests for scope extrusion.

In MetaOCaml without effects, we have implemented another solution to the counting problem in a sound staged language: Take the changed definition of *power* above. Apply our translation, then a state-passing transformation to eliminate the use of references. Finally, add staging annotations to the program so that it produces not a function of type $\mathsf{int} \to \mathsf{int}$ but a cogen of type $\langle\mathsf{int}\rangle \to \langle\mathsf{int}\rangle$.

7. Related work

Our work is inspired by the expressive staged calculus λ^{α} and its sound type system (Taha and Nielsen 2003). We aim to make the calculus model MetaOCaml more closely, by moving towards call-by-value and accounting for effects. We compare our language λ^{α}_{1v} with λ^{α} and its later development (Calcagno et al. 2004) in §3.4.

Taha (1999) poses the challenge of encoding staging using λ-abstraction in §7.2.1, 'Why lambda-abstraction is not enough for multi-stage programming'. Besides the pragmatic need to print generated code, his main reason is that escapes are hard to encode because they permit 'evaluation under lambda'. He suggests one encoding scheme, using reference cells and exceptions, which make it hard to check that the target calculus is sound. This paper answers this challenge by showing how System F with constants suffices for two-stage programming. We discuss the challenges of the encoding and our solution in §2.2–§2.4.

The safety problem of staged languages with effects has a long history. Calcagno et al. (2000) show that reference cells may safely store values of *closed types* (in our terminology, types that use no levels). Our discussion in §6 not only confirms this result, in a calculus with a small-step rather than big-step semantics, but further suggests ways to safely store open code as well. Calcagno et al. relate safe staging with effects to binding-time analyses for imperative languages. Indeed, our use of coercions is a sort of binding-time analysis that relies on the type environment in a term's derivation to describe the binding environment of the term.

Nanevski et al.'s contextual modality (2007) includes the names of free variables in the types of open code. Discussing such inclusion, Taha and Nielsen (2003; §1.4) warn of the difficulty in maintaining α-equivalence and of the need for ρ-polymorphism and negative side conditions. We avoid these nominal problems using tuple indices based on the ordering of bindings in the type environment. In contrast, Kim et al.'s staging extension to ML (2006) includes variable-capturing substitution but sacrifices α-equivalence.

Taha (2000) recounts how hard it is to develop a reduction semantics and equational theory for even an untyped staged language. He presents a confluent big-step call-by-name semantics that preserves observational equivalence for a staged language λ-U. Although that semantics has become popular (for instance used by Taha and Nielsen (2003)), we give a small-step call-by-value semantics. This choice makes formalization challenging, as our redexes may be open and our evaluation contexts may be binding, but it resolves the thorny problems described by Taha. For example, the term $(\lambda x. \langle \%x\rangle)(\lambda y. (\lambda z. z)5)$ threatens confluence under call-by-name (Taha 2000; §3.4), but our reductions are deterministic and hence trivially confluent. This term satisfies our one-level restriction and translates to $(\lambda x. \mathsf{let}\ q = \lambda(). x\ \mathsf{in}\ \lambda(). q())(\lambda y. (\lambda z. z)5)$, with no substitution conflicts or level adjustments. We encode CSP simply by enclosing an earlier-stage computation in a thunk.

Yuse and Igarashi (2006) design a staged calculus that can manipulate open code, run closed code, and persist values across stages. Their paper is rare in that it gives a call-by-value small-step semantics of a typed staged language. However, despite the clear correspondence between their calculus and linear-time temporal logic, the commuting conversions via which their 'open' and 'closed' modalities interact seem to distance their calculus from the practice of languages like MetaOCaml and thus our work with classifiers. Towards implementing their language, the authors suggest that 'designing a suitable abstract machine with environments would be a first step.' Our translation may be viewed as such a step.

8. Conclusions

Our derivation-directed translation from a staged language to an unstaged language represents future-stage code using λ-abstractions, yet preserves the evaluation order, typing, and α-equivalence (hygiene) of the staged program. This translation exposes the problem of scope extrusion as a lack of type coercion. This work prepares us to develop safe staged languages that permit combining open code fragments using state and control effects, and executing the result.

Overcoming the challenge of decoupling evaluation from scope, we have shown that call-by-value System F can encode staging, under a restriction to one future level that includes much published staged code. Lifting this restriction is the subject of the current work and may require a target language with kind polymorphism.

We have shown small-step call-by-value semantics to be a viable model of expressive staged languages such as MetaOCaml. In exchange for dealing with open redexes and binding evaluation contexts, we gain deterministic reductions and a better framework in which to account for effects, especially control.

Our translation produces much administrative overhead as it builds and uses coercion functions and environment tuples. We can either postprocess the overhead away or avoid it in the first place using a genuine staging language (like a one-pass CPS transformer).

We rely on the order of bindings in type environments to index free variables and construct coercions. For example, weakening is explicit in our translation. This reliance is reminiscent of substructural logics. Following Yuse and Igarashi (2006), it would be interesting to explicate the Curry-Howard correspondence in this regard.

Acknowledgments

We thank Olivier Danvy for the counting example in §6, Walid Taha for helpful conversations and explanations of his work, and the anonymous reviewers for helpful suggestions.

References

Ariola, Zena M., and Matthias Felleisen. 1997. The call-by-need lambda calculus. *Journal of Functional Programming* 7(3):265–301.

Asai, Kenichi. 2002. Online partial evaluation for shift and reset. In *PEPM*, 19–30.

Balat, Vincent, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 64–76.

Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2000. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP*, 25–36. LNCS 1853.

———. 2004. ML-like inference for classifiers. In *ESOP*, 79–93. LNCS 2986.

Carette, Jacques, and Oleg Kiselyov. 2005. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *GPCE*, 256–274. LNCS 3676.

Czarnecki, Krzysztof, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL implementation in MetaOCaml, Template Haskell, and C++. In *DSPG 2003*, 51–72. LNCS 3016.

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

Elliott, Conal. 2004. Programming graphics processors functionally. In *Haskell workshop*, 45–56.

Ghani, Neil, Valeria de Paiva, and Eike Ritter. 1998. Explicit substitutions for constructive necessity. In *ICALP*, 743–754. LNCS 1443.

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

Gödel, Kurt. 1933. Eine Interpretation des intuitionistischen Aussagenkalküls. *Ergebnisse eines Mathematischen Kolloquiums* 4: 39–40.

Hammond, Kevin, and Greg Michaelson. 2003. Hume: A domain-specific language for real-time embedded systems. In *GPCE*, 37–56. LNCS 2830.

Jones, Neil D. 1988. Challenging problems in partial evaluation and mixed computation. *New Generation Computing* 6(2–3): 291–302.

Kim, Ik-Soon, Kwangkeun Yi, and Cristiano Calcagno. 2006. A polymorphic modal type system for Lisp-like multi-staged languages. In *POPL*, 257–268.

Launchbury, John, and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Lisp & functional programming*, 227–238.

Lengauer, Christian, and Walid Taha, eds. 2006. *Special issue on the 1st MetaOCaml workshop (2004)*, vol. 62(1) of *Science of Computer Programming*. Elsevier.

Maraist, John, Martin Odersky, and Philip Wadler. 1998. The call-by-need lambda calculus. *Journal of Functional Programming* 8(3):275–317.

Miller, Dale A., and Alwen Tiu. 2003. A proof theory for generic judgments: An extended abstract. In *LICS*, 118–127.

Minamide, Yasuhiko, J. Gregory Morrisett, and Robert Harper. 1996. Typed closure conversion. In *POPL*, 271–283.

Nanevski, Aleksandar, Frank Pfenning, and Brigitte Pientka. 2007. Contextual modal type theory. *Transactions on Computational Logic*. In press.

Nielson, Flemming, and Hanne Riis Nielson. 1996. Multi-level lambda-calculi. In *Partial evaluation*, 338–354. LNCS 1110.

O'Hearn, Peter W., and Robert D. Tennent. 1995. Parametricity and local variables. *Journal of the ACM* 42(3):658–709.

Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP*, 157–166.

Püschel, Markus, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE special issue on program generation, optimization, and adaptation* 93(2):232–275.

Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):101–142.

Swadi, Kedar, Walid Taha, Oleg Kiselyov, and Emir Pasalic. 2006. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, 160–169.

Taha, Walid. 1999. Multi-stage programming: Its theory and applications. Ph.D. thesis, Oregon Graduate Institute of Science and Technology.

———. 2000. A sound reduction semantics for untyped CBN multi-stage computation. In *PEPM*.

———. 2004. A gentle introduction to multi-stage programming. In *DSPG 2003*, 30–50. LNCS 3016.

———. 2005. Resource-aware programming. In *ICESS*, 38–43. LNCS 3605.

Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*, 26–37.

Wright, Andrew K., and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115(1):38–94.

Yuse, Yosihiro, and Atsushi Igarashi. 2006. A modal type system for multi-level generating extensions with persistent code. In *PPDP*, 201–212.