LET (REC) INSERTION WITHOUT EFFECTS, LIGHTS OR MAGIC

OLEG KISELYOV AND JEREMY YALLOP

Tohoku University, Japan
e-mail address: oleg@okmij.org
URL: `http://okmij.org/ftp/`

University of Cambridge, UK
e-mail address: jeremy.yallop@cl.cam.ac.uk

Abstract. *Let insertion* in program generation is producing code with definitions (let-statements). Although definitions precede uses in generated code, during code generation 'uses' come first: we might not even know a definition is needed until we encounter a reoccurring expression. Definitions are thus generated 'in hindsight', which explains why this process is difficult to understand and implement – even more so for parameterized, recursive and mutually recursive definitions.

We have earlier presented an interface for *let(rec) insertion* – i.e. for generating (mutually recursive) definitions. We demonstrated its expressiveness and applications, but not its implementation, which relied on effects and compiler magic.

We now show how one can understand let insertion, and hence implement it in plain OCaml. We give the first denotational semantics of let(rec)-insertion, which does not rely on any effects at all. The formalization has guided the implementation of let(rec) insertion in the current version of MetaOCaml.

1. Introduction

Code generation, whether using quasiquotes or code combinators, is compositional: nested function calls in the generating program lead to nested expressions in the generated code, and code for larger expressions is built by incorporating code for sub-expressions unchanged. Here is an example, with code combinators (used later in the paper). Suppose `t1` denotes the code "`1+2`"; $\underline{+}$, `clet` and $\underline{\lambda}$ are the combinators to generate addition (resp., let- and lambda-expressions). Then

$\underline{\lambda}$x. (clet t1 (**fun** y $\rightarrow$ x $\underline{+}$ y))

generates the code

fun x7 $\rightarrow$ (**let** y8=1+2 **in** (x7 + y8))

Key words and phrases: metaprogramming, code generation, two-level languages, let-insertion, mutual recursion.

Preprint submitted to
Logical Methods in Computer Science

(The parentheses show the corresponding nested scopes.)

There is, however, often a need for a sub-expression to generate a let-statement that should scope over a larger (parent) expression [Kis14] – e.g. to avoid recomputations. Continuing the example, we would like to replace clet above with a combinator glet such that

$\underline{\lambda}$x. (glet t1 (**fun** y → x $\underline{+}$ y))

would now generate a more optimal

let y8=1+2 **in fun** x7 → (x7 + y8)

As this example shows, let-insertion is non-compositional: it scrambles the nesting of generated binding forms, opening the possibility of generating code with unbound or mistakenly bound variables [KKS15]. It is not a mere possibility: generating code with unbound variables does occur in practice, and is difficult to debug, as reported in [ORP16]. The non-compositionality becomes glaring when generating recursive (and especially mutually recursive) definitions [YK19].

Our aim is to understand the meaning of the let-inserting code generators, such as glet and its more general forms genlet or genletrec. We have two goals: designing a type system to statically prevent the generation of ill-scoped code; and reasoning about programs that generate let(rec) statements (not just about the code that they generate).

We report work-in-progress towards these goals: a denotational semantics that for the first time describes what genlet and genletrec mean by *themselves*, and in a compositional way. That is, the apparent intrinsic non-compositionality of let-insertion described above turns out to be a mere appearance.

The key idea is virtual let-bindings: whereas clet introduces an ordinary let-binding whose location is fixed, glet generates the code for a fresh variable accompanied by a virtual binding of that variable. Virtual bindings do not have (yet) a fixed location: they are attached to the expression that uses their bound variables and 'float up' when their attached expression is incorporated into a bigger expression. Eventually, when they reach a point that a metaprogrammer has marked with a dedicated primitive, virtual bindings are converted to real let-bindings.

Our denotational semantics is executable: it serves as a small standalone meta-programming system that implements the previously-proposed interface [YK19] for generating mutually recursive definitions; it is sufficiently complete to express the example programs used to introduce that interface. Compositionality let us build the system in pure OCaml using no effects. Furthermore, our semantics has already led to improvements and simplifications to the earlier interface.

Remark 1.1. When saying that our system performs let-insertion without any effects, we need to clarify what an effect is. After all, even lambda-abstractions [Kis17] or variable substitution [KMS21, §2.1] may be regarded as effects. Previously, let-insertion required control effects (realized as delimited control or continuation-passing transformation) [Bon92, LD94] or, at the very least, state (realized as mutable state or state-passing) [SK01]. The present paper demonstrates that none of these are needed. As a consequence, if in a generator expression such as $\mathsf{e}_1 \underline{+} \mathsf{e}_2$, the let-insertion-free summand generators e_1 and e_2 may be evaluated independently or even concurrently, they can be so evaluated even if either or both perform let-insertion. That let-insertion does not have to impose an order on evaluation is new and surprising.

Variables	x,y,z,u,f,n,r...
Types	t ::= int | bool | t → t
Expressions	e ::= x | c0 | c1 e | c2 e e | c3 e e e | λx. e | e e | **if** e **then** e **else** e | **let** x=e **in** e | **let rec** x=e **and** x=e ... **in** e
Values	v ::= c0 | λx. e

Figure 1: Syntax of the base calculus; c_i are constants of arity i: see Fig. 2, left column.

The next section introduces denotational semantics for code generation, which §3 extends to support the ordinary let insertion, and then mutually recursive-let–insertion. §4 describes the realization in the current version of MetaOCaml. Related work is briefly described in §5.

The complete code of the executable semantics along with many examples is available online.[1] MetaOCaml (version N111) incorporating let(rec) insertion is available from Opam.[2]

2. Semantics of Code Generation

As the Base calculus we take the standard call-by-value simply-typed lambda calculus with constants, ordinary let-expressions and (potentially mutually) recursive letrec-expressions: think of the most basic, side-effect–free subset of OCaml. Figure 1 presents its syntax. There, c0, c1, c2 and c3 stand for constants of the corresponding arity. The calculus includes integer and boolean literals (as zero-arity constants), the successor operation succ as an arity-1 constant, and arithmetic and comparison operations on integers, of obvious types, as arity-2 constants: see Fig. 2, left column. The type system with judgements $\Gamma \vdash$ e:t is entirely standard and elided for brevity.

A different way of presenting the calculus and its type system is in the form of an OCaml signature (Appendix A), which helps make the semantics executable.

Here are some sample expressions:

```
t1   := 1 + 2
sq   := λx. x * x
gib5 := λx.λy.
  let rec loop n =
    if n=0 then x else if n=1 then y else
    loop (n−1) + loop (n−2)
  in loop 5
```

The notation name := exp is not part of the calculus; it is used to attach a name to an expression for easy reference. The function gib5 computes the 5th element of the Fibonacci sequence whose first two elements are given as arguments.

The Base calculus both represents the code that we generate and serves as the core of the generating code. For generation, we extend Base with an additional family of types

[1] http://okmij.org/ftp/meta-programming/genletrec

[2] https://opam.ocaml.org/

0,1,2,3,... : int	$\underline{\mathsf{int}}$: int → int code
true, **false**: bool	$\underline{\mathsf{bool}}$: bool → bool code
succ: int → int	$\underline{+}$: int code → int code → int code
+: int → int → int	$\underline{=}$: int code → int code → bool code
=: int → int → bool	$\underline{\mathsf{if}}$: bool code → t code → t code → t code
	$\underline{\lambda}$: (t_1 code → t_2 code) → (t_1 → t_2) code
	$\underline{@}$: (t_1 → t_2) code → t_1 code → t_2 code
	clet: t_1 code → (t_1 code → t_2 code) → t_2 code

Figure 2: Constants of Base (left column) and its extension Codec (right column) with their types and arities: the arity of a constant is the number of arrows in its type. Although the constants may have function types, they are not expressions, unless used with the right number of arguments. The metavariable t stands for any type. For arity-2 constants such as + and =, we use infix notation. We write expressions with the constant $\underline{\lambda}$, viz., $\underline{\lambda}(\lambda \mathsf{x.e})$, as $\underline{\lambda}\mathsf{x.e}$. We silently add other arithmetic and comparison constants and code combinators, similar to + and =.

t code whose values represent generated Base expressions of type t. We also add the means of producing these code values: constants (a.k.a. code-generating combinators) in the right column of Fig.2. Below are some expressions in this extension of Base, called Codec; each expression serves as a generator of the corresponding earlier Base expression:

```
ct1  := int 1 + int 2
csq  := λx. x * x
cgib5 := λx.λy.
  let rec loop n =
    if n=0 then x else if n=1 then y else
    loop (n−1) + loop (n−2)
  in loop 5
```

Here int generates the code of an integer literal; + combines the code of summands to produce the code of an addition expression; $\underline{\lambda}$x.body generates the code of a function given a generator for its body, the variable x within the expression body representing the bound variable in the (to be) generated function.

In what sense csq and cgib5 respectively represent sq and gib5 will be clear after we describe the semantics of the calculus.

2.1. **Semantics of Base.** We consider two denotational semantics of Base, to be indexed by the subscripts R (for 'run') or S (for 'show'); X (or omitted subscript) stands for either.

First, notated by the subscript R, is the standard Scott-Strachey semantics for a typed Church-style calculus, with one small wrinkle. Its semantic domains and the interpretation $\mathcal{T}_R[\![-]\!]$ of its types are standard:

$$\mathcal{T}_R[\![\mathsf{int}]\!] = \mathbb{Z}_\perp \qquad \mathcal{T}_R[\![\mathsf{bool}]\!] = \{\mathtt{tt}, \mathtt{ff}, \perp\} \qquad \mathcal{T}_R[\![\mathsf{t_1} \to \mathsf{t_2}]\!] = \mathcal{T}_R[\![\mathsf{t_1}]\!] \to \mathcal{T}_R[\![\mathsf{t_2}]\!]$$

If A is a set and B is a domain, $A \to B$ is a continuous map from A to B, which is also a domain. We also introduce $\mathcal{N}$ as a countably infinite set of names and $\mathcal{L}$ as a set of finite sequences of small (i.e. bounded) numbers, for which we adopt the OCaml list notation.

Such a sequence can also be considered a name. Therefore, we take $\mathcal{L} \subset \mathcal{N}$ and treat $\mathcal{L}$ as names, distinct from the names appearing in source Base terms.

The semantic function $\mathcal{E}_X\big[\Gamma \vdash \mathsf{e} : \mathsf{t}\big] \in \mathcal{D}_X[\mathsf{t}]$ gives meaning to (the type derivation of) a potentially open expression e, where $\mathcal{D}_X[\mathsf{t}]$ along with auxiliary domains is defined as follows. (When writing semantic functions, we shall show only the expression rather than its entire type derivation, and often elide Γ and the type annotations to avoid clutter.)

$$\mathcal{D}_X[\mathsf{t}] = Env_X \to \mathcal{L} \to \mathcal{T}_X\big[\mathsf{t}\big]$$
$$Env_X = \mathcal{N} \rightarrowtail \mathcal{T}_X\big[\mathsf{t}\big]$$

where $A \rightarrowtail B$ is a finite map from A to B. If ρ is such a map, $\rho[k{\to}v]$ is its extension, $\rho\,|_{\neq k}$ is its restriction (removing the association for k) and $\mathrm{dom}\,\rho$ is its domain; $\varnothing$ is the empty map. We write $\mathsf{modify}\,\rho\,k\,u$ the modification of the element at key k by an update function u, that is: $\rho[k{\to}u\,\rho(k)]$.

The semantic function $\mathcal{M}_X\big[\mathsf{e} : \mathsf{t}\big] \in \mathcal{T}_X\big[\mathsf{t}\big]$ gives meaning to programs (i.e. to type derivations of closed expressions):

$$\mathcal{M}_X\big[\mathsf{e}\big] = \mathcal{E}_X\big[\mathsf{e}\big]\ \varnothing\ []$$

The semantic rules are almost entirely standard: the extra $\mathcal{L}$ argument, written as ℓ, is the wrinkle. Here is the rule for abstraction:

$$\mathcal{E}_R\big[\lambda \mathsf{x}.\mathsf{e}\big]\ \rho\ \ell = \lambda x.\mathcal{E}_R\big[\mathsf{e}\big]\ \rho[\mathsf{x}{\to}x]\ (1{::}\ell)$$

For now, ℓ is not actually used, and might as well be absent. It will help soon, in §2.2. It will also help to re-write the semantic rules in the form shown in Fig. 3, with the 'composition' and variable reference rules common to all semantics, and the semantic-specific strict mk functions. For instance, mklam_R takes the variable name and the denotation of a (generally open) expression and constructs the denotation of a lambda-abstraction. The re-written rules make it very clear that the denotation of, say, an abstraction is constructed from the denotation of the abstraction body and the name of the abstracted variable.

The semantics of Base just given could rightly be called 'extensional'. We also have an 'intensional' Base semantics, notated by the superscript S, which maps an expression to its symbolic form (a string, for example). Here $\mathcal{T}_S\big[\mathsf{t}\big]$ is always a string and the functions mklam_S etc. build strings (see Fig.3(d)). S is a trivially compositional, *bona fide* denotational semantics, and even mentioned as such by [Mos90]. Usually it is quite useless – but not here.

2.2. **Semantics of Codec.** The semantics $\mathcal{M}^X_Y[-]$ and $\mathcal{E}^X_Y\big[-\big]$ of Codec is an extension of a semantics of Base. There are now two indices: the subscript Y notates the semantics (R or S) of the generator, whereas the superscript X labels the semantics used for the generated code. (For a two-level language, Y being R is the most useful variant; we often leave this implicit and drop Y.) The semantic domain of t code is (for now; it will be extended when we come to let- and let-rec-insertion):

$$\mathcal{T}^X\big[\mathsf{t\ code}\big] = \mathcal{D}_X[\mathsf{t}]$$

A t code value represents a potentially open (think of generating function bodies) Base expression e of type t. Its meaning is, therefore, $\mathcal{E}_X[\mathsf{e}]$: the meaning the Base semantics X gives to it.

Since Codec is an extension of Base, its semantics $\mathcal{E}^X_Y\big[-\big]$ is an extension of $\mathcal{E}_Y\big[-\big]$ with the rules for the meaning of constant expressions of type t code, such as the following rules for generation of an integer literal, abstraction, application and let-expression. Generating

(a) composition rules

$$\begin{aligned}
\mathcal{E}_X\left[\mathsf{i}\right] &= \mathsf{mkint}_X\ \mathsf{i}\\
\mathcal{E}_X\left[\mathsf{e}_1 + \mathsf{e}_2\right] &= \mathsf{mkadd}_X\ \mathcal{E}_X\left[\mathsf{e}_1\right]\ \mathcal{E}_X\left[\mathsf{e}_2\right]\\
\mathcal{E}_X\left[\mathsf{x}\right] &= \mathsf{mkvar}\ \mathsf{x}\\
\mathcal{E}_X\left[\lambda\mathsf{x}.\mathsf{e}\right] &= \mathsf{mklam}_X\ \mathsf{x}\ \mathcal{E}_X\left[\mathsf{e}\right]\\
\mathcal{E}_X\left[\mathsf{e}_1\ \mathsf{e}_2\right] &= \mathsf{mkapp}_X\ \mathcal{E}_X\left[\mathsf{e}_1\right]\ \mathcal{E}_X\left[\mathsf{e}_2\right]\\
\mathcal{E}_X\left[\mathsf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2\right] &= \mathsf{mklet}_X\ \mathsf{x}\ \mathcal{E}_X\left[\mathsf{e}_1\right]\ \mathcal{E}_X\left[\mathsf{e}_2\right]
\end{aligned}$$

(b) semantics of variable reference (generic)

$$\mathsf{mkvar}\ \mathsf{x} = \lambda\rho\ell.\rho(\mathsf{x})$$

(c) mk functions for the R semantics

$$\begin{aligned}
\mathsf{mkint}_R\ i &= \lambda\rho\ell.i\\
\mathsf{mkadd}_R\ d_1\ d_2 &= \lambda\rho\ell.(d_1\,\rho\,(1::\ell)) + (d_2\,\rho\,(2::\ell))\\
\mathsf{mklam}_R\ \mathsf{v}\ d &= \lambda\rho\ell.\lambda x.d\,\rho[\mathsf{v}{\to}x]\,(1::\ell)\\
\mathsf{mkapp}_R\ d_1\ d_2 &= \lambda\rho\ell.(d_1\,\rho\,(1::\ell))\ (d_2\,\rho\,(2::\ell))\\
\mathsf{mklet}_R\ \mathsf{v}\ d_1\ d_2 &= \lambda\rho\ell.(\lambda x.d_2\,\rho[\mathsf{v}{\to}x]\,(2::\ell))\ (d_1\,\rho\,(1::\ell))
\end{aligned}$$

(d) mk functions for the S semantics

$$\begin{aligned}
\mathsf{mkint}_S\ i &= \lambda\rho\ell.\mathsf{printf"\%d"}\ i\\
\mathsf{mkadd}_S\ d_1\ d_2 &= \lambda\rho\ell.\mathsf{printf"(\%s+\%s)"}\ (d_1\,\rho\,(1::\ell))\ (d_2\,\rho\,(2::\ell))\\
\mathsf{mklam}_S\ \mathsf{v}\ d &= \lambda\rho\ell.\mathsf{printf"(\lambda\%s.\ \%s)"}\ \mathsf{v}\ (d\,\rho[\mathsf{v}{\to}\mathsf{v}]\,(1::\ell))\\
\mathsf{mkapp}_S\ d_1\ d_2 &= \lambda\rho\ell.\mathsf{printf"(\%s\ \%s)"}\ (d_1\,\rho\,(1::\ell))\ (d_2\,\rho\,(2::\ell))\\
\mathsf{mklet}_S\ \mathsf{v}\ d_1\ d_2 &= \lambda\rho\ell.\mathsf{printf"(let\ \%s{=}\%s\ in\ \%s)"}\ \mathsf{v}\\
&\qquad (d_1\,\rho\,(1::\ell))\ (d_2\,\rho[\mathsf{v}{\to}\mathsf{v}]\,(2::\ell))
\end{aligned}$$

Figure 3: Semantics of Base

addition, comparison, if-expression is similar. As far as variable references are concerned, $\mathcal{E}^X_Y\left[\mathsf{x}\right] = \mathcal{E}_Y\left[\mathsf{x}\right] = \mathsf{mkvar}\ \mathsf{x}$.

$$\begin{aligned}
\mathcal{E}^X_Y\left[\underline{\mathsf{int}}\ \mathsf{i}\right]\rho\ell &= \mathsf{mkint}_X\ \mathsf{i}\\
\mathcal{E}^X_Y\left[\underline{\lambda}\mathsf{x}.\mathsf{e}\right]\rho\ell &= \mathsf{mklam}_X\ \ell\ (\mathcal{E}^X_Y\left[\mathsf{e}\right]\rho[\mathsf{x}\to\mathsf{mkvar}\ \ell]\ (1::\ell))\\
\mathcal{E}^X_Y\left[\mathsf{e}_1\ \underline{@}\ \mathsf{e}_2\right]\rho\ell &= \mathsf{mkapp}_X\ (\mathcal{E}^X_Y\left[\mathsf{e}_1\right]\rho\,(1::l))\ (\mathcal{E}^X_Y\left[\mathsf{e}_2\right]\rho\,(2::l))\\
\mathcal{E}^X_Y\left[\mathsf{clet}\ \mathsf{e}_1\ \mathsf{e}_2\right]\rho\ell &= \mathsf{mklet}_X\ \ell\ (\mathcal{E}^X_Y\left[\mathsf{e}_1\right]\rho\,(1::l))\ ((\mathcal{E}^X_Y\left[\mathsf{e}_2\right]\rho\,(2::l))\ (\mathsf{mkvar}\ \ell))
\end{aligned}$$

When we generate an abstraction or a let-expression, the current ℓ acts as the fresh name for the (to be) bound variable. Recall that the function mklam (Fig.3) takes a variable name and the denotation for the abstraction body and gives the denotation for the abstraction. Thus the role of ℓ, the only non-standard aspect of the Base semantics, is to serve as a deterministic name generator: the fresh name to be used in an expression is the path from the root of the Böhm tree.

If we use the R semantics for the generated code (that is, choose X to be R) we see that $\mathcal{M}_R^R[\![\mathsf{ct1}]\!]$ is exactly $\mathcal{M}_R[\![\mathsf{t1}]\!]$ (which is the integer 3), $\mathcal{M}_R^R[\![\mathsf{csq}]\!]$ and $\mathcal{M}_R[\![\mathsf{sq}]\!]$ both mean the squaring function, and $\mathcal{M}_R^R[\![\mathsf{cgib5}]\!]$ and $\mathcal{M}_R[\![\mathsf{gib5}]\!]$ both mean the function that takes two arguments x and y and returns the sum of 5 copies of y and 3 copies of x.

If we use the S semantics, $\mathcal{M}_R^S[\mathsf{ct1}]$ and $\mathcal{M}_S[\mathsf{t1}]$ still coincide (both mean the string 1+2). $\mathcal{M}_R^S[\mathsf{csq}]$ and $\mathcal{M}_S[\mathsf{sq}]$ are generally different but α-equivalent lambda-expression strings. Whereas $\mathcal{M}_S[\mathsf{gib5}]$ is the string of the gib5 code (potentially α-converted), $\mathcal{M}_R^S[\mathsf{cgib5}]$ is the string

$$\lambda\mathsf{x}.\lambda\mathsf{y}.\ (((\mathsf{y}+\mathsf{x})+\mathsf{y})+(\mathsf{y}+\mathsf{x}))+((\mathsf{y}+\mathsf{x})+\mathsf{y})$$

It is an 'optimized' version of gib5, in the sense that the loop is unrolled; however, it contains several instances of code duplication. Avoiding this code duplication is where let-insertion comes in.

3. Let-insertion

To support let-insertion, we add to Codec two more forms: **let** locus l **in** e and genlet l e_m e. The former, like the ordinary let, binds the so-called locus variable l in e. In the expression genlet l e_m e, l is a locus variable (previously bound by **let** locus), e_m is a so-called memo key (for now, an int expression) and e is a t code expression (t is a Base type). Roughly, e generates the right-hand-side of the let-binding, l tells where to insert it, and the memo key instructs which e are to be shared. We describe the genlet arguments in more detail after the example, the *slightly* adjusted cgib5:

```
clgib5 := λx.λy. let locus l in
  let rec loop n =
    if n=0 then x else if n=1 then y else
    genlet l (n−1) (loop (n−1)) + genlet l (n−2) (loop (n−2))
  in loop 5
```

To a first approximation, one may think of genlet l e_m e as generating **let** z=c **in** z where z is fresh and c is the code produced by the expression e. Such 'let-expansion' is useless, however. It becomes more useful when the binding **let** z=c **in** is actually placed somewhere 'higher' in the overall generated code. The form **let** locus l marks that 'higher' place where the bindings produced by genlet are to be placed. Since let-insertion is very common, different parts of the generator may do their own let-insertions at different places; the locus variable l is to connect genlet with its corresponding let locus. Thus intuitively, genlet l e_m e will insert the **let** z=c **in** at the place marked by let locus l and return the code of the bound variable z (which is distinct from any other variables in the code).

Placing let-bindings 'higher' in the code is useful because they may be shared. The memo key defines the equivalence classes: expressions with the same memo key are to be shared. Therefore, if genlet l e_m e finds that there is already a let-binding produced by an earlier genlet with the same l and the memo key, genlet l e_m e returns the code of the earlier bound variable.

Using the semantics of these operations, explained below, we can see that whereas $\mathcal{M}_R^R[\![\mathsf{clgib5}]\!]$ remains the same as $\mathcal{M}_R[\![\mathsf{gib5}]\!]$, $\mathcal{M}_R^S[\![\mathsf{clgib5}]\!]$ is the string

λx.λy. **let** z = y **in let** u = x **in**
let v = z + u **in let** w = v + z **in**
let x6 = w + v **in** x6 + w

which is indeed an optimized version of gib5, without either loops or duplication.

3.1. **Semantics of let-insertion.** The key point was that the binding introduced by genlet | e_m e is not yet placed; its placement will be decided only later, upon seeing the corresponding **let** locus. For now, the genlet's binding is 'floating' – we say 'virtual'. To accommodate virtual bindings we extend the semantics domain of t code: it is now a tuple, whose first component is the earlier semantic domain of code values (§2.2), and whose second component is the virtual bindings. Formally, the semantic domain is:

$$\begin{aligned}\mathcal{T}^X\big[\mathsf{t\ code}\big] &= \mathcal{D}_X[\mathsf{t}] \times (\mathcal{L} \rightharpoonup \mathcal{V}^X)\\ \mathcal{V}^X &= (\mathcal{K} \times \mathcal{K})Set \times (\mathcal{K} \rightharpoonup \mathcal{B}^X)\\ \mathcal{B}^X &= \mathcal{N} \times \mathcal{D}_X[\mathsf{t}] \times \mathcal{N}Set\end{aligned}$$

Virtual bindings are indexed by the locus where they will be actually inserted. Furthermore, virtual bindings with the same locus l and the same memo key k belong to the same equivalence class. We take the locus to be an element of $\mathcal{L}$ and introduce the set $\mathcal{K}$ of memo keys. All in all, virtual bindings is a finite map $\mathcal{L} \rightharpoonup \mathcal{V}^X$, where $\mathcal{V}^X$ describes virtual bindings with the same locus. If ν is such a map, we take $\nu(l) = \varnothing$ if $l \notin \mathrm{dom}(\nu)$. Further thought, considering genlet | 2 (int 3 + genlet | 1 (int 1 + int 2)), shows that virtual bindings have to be (partially) ordered. (We shall see an example soon.) Thus the elements of $\mathcal{V}^X$ are tuples $\langle R, b\rangle$ where R is a preorder on $\mathcal{K}$ and b is a finite map $\mathcal{K} \rightharpoonup \mathcal{B}^X$. Here, $\mathcal{B}^X$ describes one equivalence class of virtual bindings: a tuple $\langle n, d, \overline{n}\rangle$ where n is the name to bind, d is the ($\mathcal{E}_X\big[-\big]$ denotation of the) expression to which n will be bound to, and $\overline{n}$ is the set of names: names equivalent to n.

The earlier semantic rules for $\mathcal{E}^X_Y\big[-\big]$ dealing with code generation have to be amended to account for the extended semantic domain: $\mathcal{E}^X_Y\big[\underline{\mathsf{int}}\ \mathsf{i}\big]$ produces the empty virtual binding, and the other rules propagate the virtual bindings of their subexpressions, merging as needed:

$$\begin{aligned}\mathcal{E}^X_Y\big[\underline{\mathsf{int}}\ \mathsf{i}\big]\rho\ell &= \langle \mathsf{mkint}_X\ \mathsf{i},\ \varnothing\rangle\\ \mathcal{E}^X_Y\big[\underline{\lambda}\mathsf{x.e}\big]\rho\ell &= \langle \mathsf{mklam}_X\ \ell\ d,\ \nu\rangle \qquad \text{where}\\ &\qquad \langle d, \nu\rangle = \mathcal{E}^X_Y\big[\mathsf{e}\big]\ \rho[\mathsf{x} \to \langle \mathsf{mkvar}\ \ell, \varnothing\rangle]\ (1 :: \ell)\\ \mathcal{E}^X_Y\big[\mathsf{e}_1\ \underline{@}\ \mathsf{e}_2\big]\rho\ell &= \langle \mathsf{mkapp}_X\ d_1\ d_2,\ \mathsf{merge}\ \nu_1\ \nu_2\rangle \qquad \text{where}\\ &\qquad \langle d_1, \nu_1\rangle = \mathcal{E}^X_Y\big[\mathsf{e}_1\big]\ \rho\,(1 :: l)\\ &\qquad \langle d_2, \nu_2\rangle = \mathcal{E}^X_Y\big[\mathsf{e}_2\big]\ \rho\,(2 :: l)\end{aligned}$$

The operation merge for virtual bindings is described later.

As we said earlier, genlet | e_m e generates a fresh name to which the code produced by e will eventually be bound; that fresh name is accompanied by the new virtual binding of that name to the result of e. Here, e is a t code expression: the generator of the expression to bind. It itself may be accompanied by virtual bindings. The new binding added by genlet | e_m e may in general depend upon those bindings, and hence has to be added as 'greater' in the preorder R. (In contrast, e_m is an int rather than an int code expression, and

so its denotation $\mathcal{E}^X_Y\left[\mathsf{e_m}\right]\rho\,(1::\ell)$, written as k below, is (when Y is R) just an integer: in general, an element of $\mathcal{K}$.)

$$\mathcal{E}^X_Y\left[\mathsf{genlet\ l\ e_m\ e}\right]\rho\ell = \langle\ \mathsf{mkvar}\,\ell,\ \ \mathsf{modify}\,\nu\,\rho(\mathsf{l})\,(\mathsf{addb}\,k\,\langle\ell, d_b\rangle)\ \rangle \qquad \text{where}$$
$$k = \mathcal{E}^X_Y\left[\mathsf{e_m}\right]\rho\,(1::\ell)$$
$$\langle d_b, \nu\rangle = \mathcal{E}^X_Y\left[\mathsf{e}\right]\rho\,(2::\ell)$$

(If e_m or e diverge, so does $\mathsf{genlet\ l\ e}_m\ \mathsf{e}$.)

The semantic function $\mathsf{addb}\,k\,\langle n, d\rangle\,v$ adds a new virtual binding of n with d to the equivalence class k of the virtual bindings $v \in \mathcal{V}^X$ with the same locus. Recall that the virtual bindings v is a pair, of preorder R and the set of equivalence classes b, indexed by the memo key. There are two cases to consider: if b already includes the equivalence class for k, we add n to that class (and disregard the right-hand-side d since we already have an equivalent one). Otherwise, we add to b the new equivalence class for k containing just the binding of n to d, and update the preorder R so that k becomes the 'latest'.

$$\mathsf{addb} : \mathcal{K} \to (\mathcal{N} \times \mathcal{D}_X[t]) \to \mathcal{V}^X \to \mathcal{V}^X$$
$$\mathsf{addb}\ k\ \langle n, d\rangle\ \langle R, b\rangle = \begin{cases} \langle R, b[k{\to}\langle n', d', \overline{n'} \cup \{n\}\rangle]\rangle \\ \qquad \text{if } b(k) = \langle n', d', \overline{n'}\rangle \\ \langle R \cup \{(k,k)\} \cup \{\langle k', k\rangle \mid k' \in \operatorname{dom} R\},\ b[k{\to}\langle n, d, \varnothing\rangle]\rangle \\ \qquad \text{if } k \notin \operatorname{dom} b \end{cases}$$

The form **let** locus l **in** e converts the virtual bindings for the locus l produced by e into real let-bindings. To a first approximation, the conversion can be understood as turning the sequence of virtual bindings into nested let-expressions. We should mind the dependency among the bindings and nest the let-expressions in the 'right' order.

$$\mathcal{E}^X_Y\left[\mathsf{let\ locus\ l\ in\ e}\right]\rho\ell = \langle \mathsf{bind}(\mathsf{ordered}\,\nu(\ell))\,d,\ \nu\mid_{\neq\ell}\rangle \qquad \text{where}$$
$$\langle d, \nu\rangle = \mathcal{E}^X_Y\left[\mathsf{e}\right]\rho[\mathsf{l}\to\ell]\,(1::\ell)$$

The semantic function $\mathsf{ordered} : \mathcal{V}^X \to \mathcal{B}^X Seq$ converts $\langle R, b\rangle$ to a sequence of bindings $\mathcal{B}^X$ in an order consistent with R. The semantic function bind converts a sequence of bindings to nested let-expressions.

$$\mathsf{bind}[\langle n_1, d_1, \overline{n_1}\rangle, \langle n_2, d_2, \overline{n_2}\rangle, \ldots]\,d =$$
$$\quad \mathsf{mklet}_X\ n_1\ d_1\ (\mathsf{subst}\,n_1\,\overline{n_1}(\mathsf{mklet}_X\ n_2\ d_2\ (\mathsf{subst}\,n_2\,\overline{n_2}(\ldots d))))$$

Recall, the semantic function mklet (see Fig.3) builds the denotation of **let** x=e **in** e' from the variable name x, $\mathcal{E}_X[\mathsf{e}]$ and $\mathcal{E}_X[\mathsf{e}']$. In a virtual binding $\langle n, d, \overline{n}\rangle$, the set $\overline{n}$ contains the variables other than n in the same equivalence class with it. They are all substituted with n:

$$\mathsf{subst}\,n\,\overline{n}\ d = \lambda\rho\ell.d\ \rho[(n'{\to}n \mid n' \in \overline{n})]\ \ell$$

The semantics function $\mathsf{merge}\,\nu_1\,\nu_2$ mentioned earlier merges the virtual bindings ν_1 and ν_2, by using addb to add one-by-one the virtual bindings of ν_2 to ν_1, in an order consistent with the R preorder of ν_2.

An example featuring code duplication and nesting should show how everything fits together. As a warm-up, a generator without genlet:

```
let locus l in
 let x = (int 6 + int 7) in
 ((x + int 20) * (x + int 30)) / int 100
```

has the meaning in the $\mathcal{M}_R^S[-]$ semantics as a string

$$(((6 + 7) + 20) * ((6 + 7) + 30)) \;/\; 100$$

with evidently duplicate code. We can eliminate the duplication by putting let-expressions into the generated code, using genlet:

```
1  let locus l in
2   let x = genlet l 1
3               (int 6 + int 7) in
4   (genlet l 2 (x + int 20)
5     *
6     genlet l 3 (x + int 30))
7    / int 100
```

The three genlet expressions are each in their own equivalence class and hence pass distinct memo keys as their second arguments. (Such usage can be automated – in fact, it has been, in MetaOCaml.) The code is typeset so that each notable expression is on its own line for easy reference. Furthermore, we write ℓ_i for the Böhm tree location (an element of $\mathcal{L}$) corresponding to line i.

The denotation is computed as follows. Let ρ_0 be the initial environment and $\rho = \rho_0[\mathsf{l} \to \ell_1]$. We also assume mkmul and mkdiv functions similar to mkadd.

$$\mathcal{E}^X\big[(\underline{\mathsf{int}}\,6 \mathbin{\underline{+}} \underline{\mathsf{int}}\,7)\big]\rho\ell_3 = \langle d_3,\ \varnothing\rangle \qquad \text{where } d_3 = \mathsf{mkadd}_X\,(\mathsf{mkint}_X\,6)\,(\mathsf{mkint}_X\,7)$$

$$\mathcal{E}^X\big[\mathsf{genlet}\,\mathsf{l}\,1\,(\underline{\mathsf{int}}\,6 \mathbin{\underline{+}} \underline{\mathsf{int}}\,7)\big]\rho\ell_2 = \langle \mathsf{mkvar}_X\,\ell_2,\ \{\ell_1 \to v_2\}\rangle \quad \text{where}$$
$$v_2 = \mathsf{addb}\,1\langle \ell_2, d_3\rangle\,\varnothing = \langle\{(1,1)\}, \{1 \to \langle \ell_2, d_3, \varnothing\rangle\}\rangle$$

This was the denotation of the expression the variable x is bound to. The variable is then used twice, on lines 4 and 6. In the following we take ρ_1 to be ρ extended with the binding for x.

$$\mathcal{E}^X\big[\mathsf{genlet}\,\mathsf{l}\,2\,(\mathsf{x} \mathbin{\underline{+}} \underline{\mathsf{int}}\,20)\big]\rho_1\ell_4 = \langle \mathsf{mkvar}_X\,\ell_4,\ \{\ell_1 \to v_4\}\rangle \quad \text{where}$$
$$d_4 = \mathsf{mkadd}_X\ (\mathsf{mkvar}_X\,\ell_2)\ (\mathsf{mkint}_X\,20)$$
$$v_4 = \mathsf{addb}\,2\,\langle \ell_4, d_4\rangle\,v_2 = \langle\{(2,2),(1,1),(1,2)\}, \{1 \to \langle \ell_2, d_3, \varnothing\rangle, 2 \to \langle \ell_4, d_4, \varnothing\rangle\}\rangle$$

$$\mathcal{E}^X\big[\mathsf{genlet}\,\mathsf{l}\,3\,(\mathsf{x} \mathbin{\underline{+}} \underline{\mathsf{int}}\,30)\big]\rho_1\ell_6 = \langle \mathsf{mkvar}_X\,\ell_6,\ \{\ell_1 \to v_6\}\rangle \quad \text{where}$$
$$d_6 = \mathsf{mkadd}_X\ (\mathsf{mkvar}_X\,\ell_2)\ (\mathsf{mkint}_X\,30)$$
$$v_6 = \mathsf{addb}\,3\,\langle \ell_6, d_6\rangle\,v_2 = \langle\{(3,3),(1,1),(1,3)\}, \{1 \to \langle \ell_2, d_3, \varnothing\rangle, 3 \to \langle \ell_6, d_6, \varnothing\rangle\}\rangle$$

When computing the denotation for the product expression, we have to merge the virtual bindings v_4 and v_6

$$\mathcal{E}^X\big[\ldots \mathbin{\underline{*}} \ldots\big]\rho_1\ell_5 = \langle \mathsf{mkmul}_X\,(\mathsf{mkvar}_X\,\ell_4)\,(\mathsf{mkvar}_X\,\ell_6),\ \{\ell_1 \to v_5\}\rangle \quad \text{where}$$
$$v_5 = \langle\{(3,3),(2,2),(1,1),(1,3),(1,2),(2,3)\},$$
$$\{1 \to \langle \ell_2, d_3, \varnothing\rangle, 2 \to \langle \ell_4, d_4, \varnothing\rangle, 3 \to \langle \ell_6, d_6, \varnothing\rangle\}\rangle$$

The merged virtual bindings propagate to the denotation of the division expression, and converted to real let-bindings by let-locus. The whole expression thus has the denotation

$$\begin{array}{l}\mathcal{E}^X\big[\mathsf{let\ locus\ l\ in\ \ldots}\big]\rho_o\ell_1 = \langle\\ \quad \mathsf{mklet}_X\,\ell_2\,(\mathsf{mkadd}_X\,(\mathsf{mkint}_X\,6)\,(\mathsf{mkint}_X\,7))\ (\\ \quad \mathsf{mklet}_X\,\ell_4\,(\mathsf{mkadd}_X\ (\mathsf{mkvar}_X\,\ell_2)\ (\mathsf{mkint}_X\,20))\ (\\ \quad \mathsf{mklet}_X\,\ell_6\,(\mathsf{mkadd}_X\ (\mathsf{mkvar}_X\,\ell_2)\ (\mathsf{mkint}_X\,30))\ (\\ \quad\quad \mathsf{mkdiv}_X\ (\mathsf{mkmul}_X\,(\mathsf{mkvar}_X\,\ell_4)\,(\mathsf{mkvar}_X\,\ell_6))\ (\mathsf{mkint}_X\,100)))),\ \varnothing\rangle\end{array}$$

The interface for genlet described above differs from our previous proposal [YK19] in that here we combine memoization and let-insertion. Although both memoization and let-insertion are usually implemented in terms of effects, we have used no effects at all.

If **let** locus in clgib5 is positioned above $\underline{\lambda}$y. . . ., so called scope-extrusion occurs, resulting in the generated code with have unbound variables (as we can verify in our semantics). It is the subject of ongoing work to develop a type system to statically prevent such problems.

3.2. **Generating (mutually) recursive definitions.** It turns out that genletrec for generating (mutually) recursive definitions presented by [YK19] is a minor variant of the above genlet, with almost the same semantics. The syntax is the same: genletrec l e_m e for requesting the binding (and obtaining the name of the to-be-bound variable) and **let rec** locus l **in** e for actually generating let-rec statements at that locus.

Here is the example: specializing the Ackermann function challenge by Neil Jones. The ack2 below is the two-argument Ackermann function ack partially applied to two:

```
ack2 :=
  let rec ack = λm.λn.
    if m=0 then n+1 else
    if n=0 then ack (m−1) 1 else
    ack (m−1) (ack m (n−1))
  in ack 2
```

Below is the generator of ack2, matching ack2 in form.

```
cack2 :=
  let rec locus l in
  let rec ack = λm.λ̲ n.
    if m=0 then n+̲ (int̲ 1) else
    if̲ (n =̲ int̲ 0)
      (genletrec l (m−1) (ack (m−1)) @̲ int̲ 1)
      (genletrec l (m−1) (ack (m−1)) @̲ (genletrec l m (ack m) @̲ (n−̲int̲ 1)))
  in genletrec l 2 (ack 2)
```

Here genletrec is needed not just to obtain optimal code; without genletrec we get no code at all: divergence. Using the semantics below, $\mathcal{M}^S_R\big[\mathsf{cack2}\big]$ gives the following code

```
let rec x = λu. if u = 0 then y 1 else y (x (u − 1))
and y = λv. if v = 0 then z 1 else z (y (v − 1))
and z = λw. w + 1 in x
```

Semantically, genletrec is also close to genlet: both return the name of a to-be-bound variable, accompanied by a virtual binding. Whereas for genlet l e_m m the binding associates

the name to the expression produced by e (the generated code), for $\mathsf{genletrec\ |\ e}_m\ \mathsf{e}$ the virtual binding associates the name to e itself (one may say, unevaluated generator expression). Thus genletrec is even lazier, introducing even more virtual bindings. Formally, we extend $\mathcal{B}^X$ to

$$\mathcal{B}'^X = \mathcal{N} \times (\mathcal{D}_X[\mathsf{t}] + \mathcal{T}^X\big[\mathsf{t\ code}\big]) \times \mathcal{N}Set$$

where $A + B$ is to be read as a disjoint union (with tags `inl` and `inr`) with a separately added $\bot$. The left summand $\mathcal{D}_X[\mathsf{t}]$ is inherited from $\mathcal{B}^X$: the meaning of the code for the right-hand-side of the binding. The right summand $\mathcal{T}^X\big[\mathsf{t\ code}\big]$ is (the approximation of) the code for the right-hand-side. The $\mathcal{T}^X\big[\mathsf{t\ code}\big]$ definition hence becomes recursive and has to be understood as the solution to the domain equation. Then

$$\mathcal{E}^X_Y\big[\mathsf{genletrec\ |\ e_m\ e}\big]\rho\ell = \langle \mathsf{mkvar}\,\ell,\ \varnothing[l{\to}\mathsf{addb}\,k\,\langle \ell, \mathtt{inr}\ \mathcal{E}^X_Y\big[\mathsf{e}\big]\,\rho\,(2 :: \ell)\rangle\,\varnothing]\rangle \qquad \text{where}$$
$$l = \rho(\mathsf{l})$$
$$k = \mathcal{E}^X_Y\big[\mathsf{e_m}\big]\ \rho\,(1 :: \ell)$$

Virtual bindings now contain yet to be evaluated generator expressions for the right-hand-side of bindings. To generate the let-rec-statement, we have to evaluate them – which may create more virtual bindings, which have to be merged and again evaluated (the evaluation may also diverge). This complex process of evaluation is called canonicalization, and performed by the semantic function $\mathsf{canon}\,\nu\,l$ which takes virtual bindings and the locus l and returns updated bindings ν' such that $\nu'(l)$ are all canonical: each $\mathcal{B}'^X$ is actually $\mathcal{B}^X$. The function is the least fixpoint of the following recursive equation.

$$\mathsf{canon}\,\nu\,l = \begin{cases} \nu \text{ if } \nu(l) \text{ are all canonical} \\ \mathsf{canon}\,(\mathsf{merge}\ \nu[l{\to}\langle R, b'\rangle]\ \nu'')\,l \text{ where} \\ \qquad \langle R, b\rangle = \nu(l) \\ \qquad \langle n, \mathtt{inr}\ d', \overline{n}\rangle = b(k), \quad k \in \mathrm{dom}(b) \\ \qquad \langle d, \nu''\rangle = d' \\ \qquad b' = b[k{\to}\langle n, \mathtt{inl}\ d, \overline{n}\rangle] \end{cases}$$

That is, among the bindings $\nu(l)$ with the same locus l we pick an equivalence class with a non-canonical virtual binding $\langle n, \mathtt{inr}\ d', \overline{n}\rangle$. If such a class does not exist, we are done. Let k be the memo key of that class. If d' is $\bot$, so is the whole $\mathsf{canon}\,\nu\,l$. If not, it is a pair, containing (the meaning of) the code d for the right-hand-side of the binding, plus its accompanying virtual bindings ν''. We update the class k so it now contains the canonical binding $\langle n, \mathtt{inl}\ d, \overline{n}\rangle$ and merge the result with ν''. It may happen that ν'' contains a non-canonical binding for the same locus l and the same memo key k. In fact, this happens in the cack2 example above: evaluating ack 2 produces virtual bindings that contain ack 2 again. The operation merge folds such bindings, with the already canonical `inl` d used as the right-hand-side for the $\nu''(l)(k)$ binding – thus canonicalization may eventually terminate.

The form **let rec** locus | **in** e is semantically almost the same as **let** locus | **in** e, differing only in the extra step of canonicalization of the virtual bindings produced by e. After canonicalization, the virtual bindings are converted into real bindings in the same way as for **let** locus | **in** e (only we produce one letrec-expression rather than a nest of let-expressions, and therefore do not bother with ordered.)

4. MetaOCaml Implementation

Formal semantics is not the end, but the means; it is developed to be used. One application, the subject of future work, is reasoning about generating programs and making sure the generated code is not only well-formed and well-typed but also intended. Another application, the subject of the present paper, is to clarify edge cases, and attempt to minimize them once exposed. The formal development has already proven useful: it has improved our previous design for generating mutually recursive definitions [YK19], which led to the straightforward implementation in the current (N111) version of MetaOCaml [Kis14]. We now briefly describe the implemented interface.

The let-insertion interface in MetaOCaml is as follows:

```
type locus
val locus_global : locus
val genlet : ?name:string → ?locus:locus → α code → α code
val with_locus : (locus → ω code) → ω code

type locus_rec
val mkgenlet : ?name:string → locus_rec → (κ→κ→bool) →
  ((κ → (α→β) code) → (κ → (α→β) code))
val with_locus_rec : (locus_rec → ω code) → ω code
```

Here genlet is a version of genlet ! e_m e described earlier, for the case of all memo keys being distinct. The first optional argument of the MetaOCaml genlet is the hint for the variable name to generate (useful if one wishes to see variable names in the generated code other than t_34 and the like.) Locus is the second argument: if omitted, it defaults to locus_global, which is the implicit locus at the very beginning of the program. The current interface (and also the implementation) thus subsumes the locus-less genlet of the previous MetaOCaml version.

The type locus_rec and the operations mkgenlet and with_locus_rec deal with potentially mutually recursive memoizing let-insertion. In the earlier cack2 we have seen that the memoization key occurs also in the expression to bind, as an argument to some function. Such a pattern seems common, and mkgenlet interface is built around it. Furthermore, memo keys are no longer integers; therefore, the user has to supply the key comparison function κ→κ→bool. Again, it is better to see an example – the same Ackermann function specialization example, but written this time in MetaOCaml (the example is part of the MetaOCaml test suite).

```
let sack m =
  with_locus_rec @@ fun l →
  let g = mkgenlet l (=) in
  let rec loop m =
    if m = 0 then .<fun n → n + 1>. else
    .<fun n → if n = 0 then .~(g loop (m−1)) 1
       else .~(g loop (m−1)) (.~(g loop m) (n−1))>.
  in g loop m
```

The interface is designed so that we could blindly put g before every recursive call, and obtained the desired generator. The generated code is identical to that for cack2 shown earlier.

The major difference of MetaOCaml let-insertion is that it never produces ill-scoped code (that is, code with a scope extrusion). In MetaOCaml, let(rec) is actually inserted either at the place of the explicit **let** locus, or at the binding that dominates all free variables of the bound expression, whichever has the narrowest scope.

5. Related Work

It was recognized early on [Bon92, LD94] that one can use control effects (either direct or realized via CPS) to answer the compositionality challenge of the ordinary, well-nested let-insertion. [KKS11] give a comprehensive formal treatment. Unfortunately, neither the standard CPS nor the well-understood shift operator are of any help with let-insertion that does not follow the stack discipline and crosses already-generated bindings. This problem was discussed in [KKS15], which proposed a very complicated transformation for hygienic let-insertion across bindings, whose correctness was only conjectured. The theory of code movement across already-generated bindings was later developed in [KKS16], using operational semantics; it did not include let-insertion however.

Semantics for multi-stage languages have from the very earliest works [TBS98] generally been given operationally; our denotational presentation is unusual in this respect. In contrast, earlier work on two-level languages used a similar style to ours: [NN92] give a denotational semantics in which the meta-language semantics is parameterized by the semantics of the base language. (However, that work did not investigate let-insertion.)

Generating (mutually) recursive bindings has not previously been formally considered at all, to our knowledge.

6. Conclusions

We have developed an executable denotational semantics for let(rec) insertion. The next step is to develop a type system that prevents scope extrusion. Our semantics, for the first time, lets us reason about the code with the generated let-statements, and we plan to demonstrate this facility on standard interesting examples (e.g. from [KKS11, YK19, KKS16]).

Acknowledgments

We are grateful to the anonymous reviewers for many very helpful suggestions, which greatly improved the presentation. We thank Yukiyoshi Kameyama for hospitality. This work was partially supported by JSPS KAKENHI Grant Number 18H03218.

References

[Bon92] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 1–10, New York, NY, USA, 1992. ACM.

[Kis14] Oleg Kiselyov. The design and implementation of BER MetaOCaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer International Publishing, 2014.

[Kis17] Oleg Kiselyov. Higher-order programming is an effect, 2017. HOPE Workshop at ICFP 2017.

[KKS11] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: Staging with delimited control. *Journal of Functional Programming*, 21(6):617–662, November 2011.

[KKS15] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Combinators for impure yet hygienic code generation. *Science of Computer Programming*, 112 (part 2):120–144, November 2015.

[KKS16] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers - type- and scope-safe code generation with mutable cells. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 271–291, 2016.

[KMS21] Oleg Kiselyov, Shin-Cheng Mu, and Amr Sabry. Not by equations alone: Reasoning with extensible effects. *Journal of Functional Programming*, 31:e2, 2021.

[KS16] Oleg Kiselyov and K. C. Sivaramakrishnan. Eff directly in OCaml. In Kenichi Asai and Mark R. Shinwell, editors, *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016*, volume 285 of *EPTCS*, pages 23–58, 2016.

[LD94] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 227–238, New York, 1994. ACM.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, NY, 1990.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, 1992.

[ORP16] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. RandIR: differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016*, pages 21–30. ACM, October 30 - November 4 2016.

[SK01] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2–3):101–142, September 2001.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929. Springer, 1998.

[YK19] Jeremy Yallop and Oleg Kiselyov. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pages 75–81, New York, NY, USA, 2019. ACM.

Appendix A. Base and Codec, executable

The base calculus can be represented as an OCaml signature. Calculus expressions of type α are represented as OCaml values of type α repr. The mutually recursive mletrec takes a collection of clauses indexed by idx; the first argument to mletrec indicates the number of clauses. We gave mletrec a general type, with polymorphic α. In all practical cases, α should be a function type and $(\alpha{\to}\beta)$ repr should start with a lambda. The form letrec is an instance of the general mletrec.

The R and S semantics of the calculus are then implementations of the signature. Using OCaml to express the denotational semantics was argued in [KS16, §3.1.4].

```
type α repr

val lam  : (α repr → β repr) → (α→β) repr
val let_ : α repr → (α repr →β repr) → β repr
val (/)  : (α→β) repr → (α repr → β repr) (* application *)
val if_  : bool repr → α repr → α repr → α repr

type idx = int
val mletrec : idx →
  ((idx → α repr) → (idx → α repr)) →
  ((idx → α repr) → ω repr) → ω repr

val letrec : ((α→β) repr → α repr → β repr) →
     ((α→β) repr → ω repr) → ω repr

val int  : int → int repr
val bool : bool → bool repr

val succ  : int repr → int repr
val ( +  ) : int repr → int repr → int repr
val ( −  ) : int repr → int repr → int repr
val ( ∗  ) : int repr → int repr → int repr
val ( =. ) : int repr → int repr → bool repr
```

Figure 4: Base calculus represented in OCaml: its syntax as OCaml signature

```
type α cd  (* the type of code values *)

val clam  : (α cd repr → β cd repr) → (α→β) cd repr
val clet : α cd repr → (α cd repr → β cd repr) → β cd repr

val int : int  → int cd repr
val bool : bool → bool cd repr

val csucc : int cd repr → int cd repr

val ( + ) : int cd repr → int cd repr → int cd repr
val ( - ) : int cd repr → int cd repr → int cd repr
val ( * ) : int cd repr → int cd repr → int cd repr
val (=) : int cd repr → int cd repr → bool cd repr

val (/) : (α → β) cd repr → α cd repr → β cd repr
val if_  : bool cd repr → α cd repr → α cd repr → α cd repr

type memo_key = int repr
type locus_t
val genlet_locus : (locus_t → ω cd repr) → ω cd repr
val genlet : locus_t → memo_key → α cd repr → α cd repr

val genletrec_locus : (locus_t → ω cd repr) → ω cd repr
val genletrec : locus_t → memo_key → (α→β) cd repr → (α→β) cd repr
```

Figure 5: Codec calculus represented in OCaml: its syntax as OCaml signature. The calculus includes the whole Base calculus; only the extension is shown.

Figure 5 presents the Codec calculus in the form of an OCaml signature. To be precise, the figure shows only the extension of Base, with the type of code values and combinators to produce them. A code value is annotated only with the type of the expression it generates, with no further classifiers (at present).

The combinator clam builds a lambda-expression given the open expression for the function body. Let-insertion combinators are the simple, local let-insertion clet demonstrated in §1; genlet for non-recursive definitions, described in §3 and genletrec for mutually recursive definitions, §3.2. The locus argument of the latter two is a variable bound by the corresponding genlet_locus, not an expression; therefore its type is just locus_t rather than locus_t repr.