let (rec) insertion without effects, lights or magic

Oleg Kiselyov
Tohoku University, Japan

Jeremy Yallop
University of Cambridge, UK

Abstract

At last year's ML Family Workshop we 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.

1 Summary

Code generation, whether using quasiquotes or code comibinators, 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. There is, however, often a need for a sub-expression to generate a let-statement that should scope over a larger (parent) expression [Kiselyov 2014] – e.g. to avoid recomputations. The non-compositionality becomes glaring when generating recursive, especially mutually recursive definitions [Yallop and Kiselyov 2019]. Non-compositionality of let-insertion scrambles the nesting of generated binding forms – which opens the all too real possibility of generating code with unbound or mistakenly bound variables.

Our goal is to understand the meaning of the let-inserting code generators, such as genlet or genletrec. We have two aims: designing a type system to statically prevent producing 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. Our denotational semantics is executable: it serves as a small standalone meta-programming system that implements the previously-proposed interface 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 whatsoever (neither control effects nor even state, and without state-passing or CPS). Furthermore, our semantics has already led to improvements and simplifications to the proposed interface.

The code is available online:
http://okmij.org/ftp/meta-programming/genletrec

2 Introduction

To develop intuitions and save space, we avoid a formal presentation and instead use examples to introduce code generation with let(rec)-insertion and its meaning. (Some formalities can be found in the Appendix.) This section introduces a semantics for code generation; §3 extends the calculus to support let insertion.

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. Here are some sample expressions:

$$
\begin{aligned}
&\mathsf{t1} \;:= 1 + 2\\
&\mathsf{sq} \;:= \lambda \mathsf{x}.\ \mathsf{x} * \mathsf{x}\\
&\mathsf{gib5} := \lambda \mathsf{x}.\lambda \mathsf{y}.\textbf{let rec}\ \mathsf{loop\ n} =\\
&\qquad\qquad \textbf{if}\ \mathsf{n{=}0}\ \textbf{then}\ \mathsf{x}\ \textbf{else if}\ \mathsf{n{=}1}\ \textbf{then}\ \mathsf{y}\ \textbf{else}\\
&\qquad\qquad \mathsf{loop\ (n-1) +\ loop\ (n-2)}\ \textbf{in}\ \mathsf{loop\ 5}
\end{aligned}
$$

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 t code whose values represent generated Base expressions of type t – and with a means of producing these code values. Below are some expressions in this extension of Base, called Codec; each expression serves as a generator of the corresponding earlier Base expression:

$$
\begin{aligned}
&\mathsf{ct1} \;:= \underline{\mathsf{int}}\ 1\ \underline{+}\ \underline{\mathsf{int}}\ 2\\
&\mathsf{csq} \;:= \underline{\lambda} \mathsf{x}.\ \mathsf{x}\ \underline{*}\ \mathsf{x}\\
&\mathsf{cgib5} := \underline{\lambda} \mathsf{x}.\underline{\lambda} \mathsf{y}.\ \textbf{let rec}\ \mathsf{loop\ n} =\\
&\qquad\qquad \textbf{if}\ \mathsf{n{=}0}\ \textbf{then}\ \mathsf{x}\ \textbf{else if}\ \mathsf{n{=}1}\ \textbf{then}\ \mathsf{y}\ \textbf{else}\\
&\qquad\qquad \mathsf{loop\ (n-1)}\ \underline{+}\ \mathsf{loop\ (n-2)}\ \textbf{in}\ \mathsf{loop\ 5}
\end{aligned}
$$

Here $\underline{\mathsf{int}}$ of the type int → int code generates the code of an integer literal; $\underline{+}$ of the type int code → int code → int code combines the code of summands to the code of the 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 represents the bound variable in the (to be) generated function.

In what sense csq and cgib5 represent sq (resp. gib5) should be clear after we describe the semantics of the calculi. We consider two denotational semantics of Base; both come with their own sets of type-indexed semantic domains V_t and with two semantic functions: $\mathcal{M}[\![-]\!]$, for the meaning of whole programs, and $\mathcal{E}[\![-]\!]$, for the meaning of potentially open expressions.

The first denotational semantics, notated by the subscript R, is the standard Scott-Strachey semantics for a typed Church-style calculus, with one small wrinkle. V_t are the standard lifted domains (e.g., V_{int} is the set of integers with ⊥). We assume another semantic domain V_{nom} whose elements are finite sequences of small numbers (for which we adopt the OCaml list notation). There clearly is a bijection between V_{nom} and variable names. Therefore, we shall refer to the elements of V_{nom} as names (distinct from the names appearing in source Base terms) and use them as such. The semantic function $\mathcal{M}_R[\![\mathsf{e{:}t}]\!]$ maps (a type derivation of a closed) expression e of type t to V_t; $\mathcal{E}_R[\![\Gamma \vdash \mathsf{e{:}t}]\!]$ maps e to a continuous function from the environment and V_{nom} to V_t. The extra V_{nom} argument, written as ℓ, is the wrinkle. (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.)

The environment ρ is a strict finite map from variable names to V_t; the environment extension is written $\rho[\mathsf{x}\longrightarrow\mathsf{v}]$. The semantic rules are entirely standard (modulo ℓ). We show only the rules for abstraction and application:

$$\mathcal{E}_R[\![\lambda \mathsf{x.e}]\!]\ \rho\ \ell = \lambda x.\ \mathcal{E}_R[\![\mathsf{e}]\!]\ \rho[\mathsf{x}\longrightarrow x]\ (1{::}\ell)$$
$$\mathcal{E}_R[\![\mathsf{e}_1\ \mathsf{e}_2]\!]\ \rho\ \ell = (\mathcal{E}_R[\![\mathsf{e}_1]\!]\rho\ (1{::}\ell))\ (\mathcal{E}_R[\![\mathsf{e}_2]\!]\rho\ (2{::}\ell))$$
$$\mathcal{M}_R[\![\mathsf{e}]\!] = \mathcal{E}_R[\![\mathsf{e}]\!]\ \varnothing\ []$$

For now, ℓ is not actually used, and might as well be absent. It will help later. It will also help to re-write the semantic rule for abstraction in the following form:

$$\mathcal{E}_R[\![\lambda \mathsf{x.e}]\!] = \mathsf{mkl}_R\ \mathsf{x}\ \mathcal{E}_R[\![\mathsf{e}]\!]$$
$$\text{where } \mathsf{mkl}_R\ \mathsf{v}\ d = \lambda\rho\ell x.\ d\ \rho[\mathsf{v}\longrightarrow x]\ (1{::}\ell)$$

where the semantic function mkl_R takes the variable name and the denotation of a (generally open) expression and constructs the denotation of a lambda-abstraction. The re-written rule makes it very clear that the denotation of 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 show an 'intensional' Base semantics, notated by the superscript S, which maps an expression to its symbolic form (a string, for example). Here every element of V_t is always a string, regardless of t. It is a trivially compositional, *bona fide* denotational semantics, and even mentioned as such by Mosses [1990]. Usually it is quite useless – but not here.

The semantics of $\mathcal{E}^C[\![-]\!]$ of Codec is an extension of the R semantics of Base[1]. Since what we generate are (potentially open: think of generating function bodies) Base expressions e of type t, the meaning of a t code value is $\mathcal{E}[\![\mathsf{e}]\!]$: the meaning of e according to R or S or some other semantics of Base[2]. We pair $\mathcal{E}[\![\mathsf{e}]\!]$ with a sequence of so-called virtual bindings v detailed in the next section; they can be disregarded for now. The following are two sample cases for the semantic function $\mathcal{E}^C[\![-]\!]$ of Codec, for generating an integer literal and an abstraction.

$$\mathcal{E}^C[\![\underline{\mathsf{int}}\ \mathsf{i}]\!]\ \rho\ \ell = (\mathcal{E}[\![\mathsf{i}]\!],\varnothing)$$
$$\mathcal{E}^C[\![\underline{\lambda}\mathsf{x.e}]\!]\ \rho\ \ell = \mathsf{map}_1\ (\mathsf{mkl}\ \ell)\ (\mathcal{E}^C[\![\mathsf{e}]\!]\ \rho[\mathsf{x}\longrightarrow(\mathcal{E}[\![\ell]\!],\varnothing)]\ (1{::}\ell))$$
$$\text{where } \mathsf{map}_1\ f\ (d, v) = (f d, v)$$

When we generate an abstraction, the current ℓ acts as the fresh name for the (to be) bound variable. Recall that the function mkl provided by a Base semantics takes a variable name and the denotation for the abstraction body and gives the denotation for the abstraction.

If we use the R semantics for the generated code (that is, choose $\mathcal{E}[\![-]\!]$ to be $\mathcal{E}_R[\![-]\!]$) we see that $\mathcal{M}^C_R[\![\mathsf{ct1}]\!]$ is exactly $\mathcal{M}_R[\![\mathsf{t1}]\!]$ (which is the integer 3), $\mathcal{M}^C_R[\![\mathsf{csq}]\!]$ and $\mathcal{M}_R[\![\mathsf{sq}]\!]$ both mean the squaring function, and $\mathcal{M}^C_R[\![\mathsf{cgib5}]\!]$ 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, $\mathcal{M}^C_S[\![\mathsf{ct1}]\!]$ and $\mathcal{M}_S[\![\mathsf{t1}]\!]$ still coincide (both mean the string 1+ 2). $\mathcal{M}^C_S[\![\mathsf{csq}]\!]$ and $\mathcal{M}_S[\![\mathsf{sq}]\!]$ are generally different but alpha-equivalent lambda-expression strings. $\mathcal{M}^C_S[\![\mathsf{cgib5}]\!]$ is now the string

λx.λy. (((y + x) + y) + (y + x)) + ((y + x) + 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, similarly to 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 (which we take for now to be an int code expression). It is better explained by example, of the *slightly* adjusted cgib5:

clgib5 := $\underline{\lambda}$x.$\underline{\lambda}$y. **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)) $\underline{+}$ 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 thus 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 may be shared. Therefore, if genlet l e_m e finds that there is already a let-binding produced by an earlier genlet l e_m' e' 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 $\mathcal{M}^C_R[\![\mathsf{clgib5}]\!]$ remains the same as $\mathcal{E}_R[\![\mathsf{gib5}]\!]$, $\mathcal{M}^C_S[\![\mathsf{clgib5}]\!]$ 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.

In the semantics of let-insertion forms, we take the locus l to be an element of V_{nom} and introduce 'virtual bindings' v as sequences of tuples (l, k, n, e_b) with $\varnothing$ for the empty sequence and $\cdot$ for the element- or sequence concatenation. Each tuple in v represents one let-binding, *not yet generated*: l is a locus (an element of V_{nom}), k is a memo key (an integer), and n and e_b represent the binding: $n \in V_{nom}$ is the variable to be bound, and e_b is $\mathcal{E}[\![\mathsf{e}]\!]$, the meaning of the expression e that n will be bound to.

The semantic rules are as follows:

$$\mathcal{E}^C[\![\mathsf{genlet\ l\ e}_m\ \mathsf{e}]\!]\ \rho\ \ell = (\mathcal{E}[\![\ell]\!],\ v\cdot(l, k, \ell, e_b))$$
where

[1] For a two-level language, the S semantics for Codec in unnecessary.

[2] The absence of the subscript in $\mathcal{E}$ means we are talking about any semantics.

$$l = \rho(\mathsf{l})$$
$$k = \mathcal{E}^C[\![\mathsf{e}_m]\!]\ \rho\ (1{::}\ell)$$
$$(e_b, v) = \mathcal{E}^C[\![\mathsf{e}]\!]\ \rho\ (2{::}\ell)$$

(Note that e_m should be an integer expression, and hence its meaning (given ρ and ℓ) is an integer. On the other hand, e is assumed to be an expression of the code type, whose meaning is the tuple: the meaning of the produced code plus the virtual bindings.)

$$\mathcal{E}^C[\![\textbf{let locus } \mathsf{l} \textbf{ in } \mathsf{e}]\!]\ \rho\ \ell = ((\mathsf{bg\ g1} \ldots (\mathsf{bg\ gn}\ d)), v_{\neq\ell})$$
where
$$(d,v) = \mathcal{E}^C[\![\mathsf{e}]\!]\rho[\mathsf{l}{\to}\ell]\ (1{::}\ell)$$
$$[\mathsf{g1},\ldots,\mathsf{gn}] = \mathsf{groupby\ k}\ v_{=\ell}$$
$$\mathsf{bg}\ [(l,k,n,\mathcal{E}[\![\mathsf{e}]\!]),\ (\_,\_,n_1,\_),\ldots]\ \mathcal{E}[\![\mathsf{e'}]\!] =$$
$$\quad \mathsf{mklet}\ n\ \mathcal{E}[\![\mathsf{e}]\!]\ (\mathsf{subst}\ [n_1{\to}n,\ldots]\ \mathcal{E}[\![\mathsf{e'}]\!])$$

Here, $v_{=\ell}$ and $v_{\neq\ell}$ are the partitions of v into the sequences of bindings whose locus is ℓ (resp., not ℓ). The semantic function mklet, like the mkl seen earlier, builds the denotation of **let** x=e **in** e' from the variable name x, $\mathcal{E}[\![\mathsf{e}]\!]$ and $\mathcal{E}[\![\mathsf{e'}]\!]$.

Intuitively, genlet l e_m e produces a virtual (floating) let-binding: it means the code for a fresh name, annotated with the code of the expression it will be bound to (when the time comes to actually generate the let-expression code). On the other hand, **let locus** l **in** e converts the virtual bindings in the code generated by e into real let-bindings; to be precise, only the bindings annotated with the l's locus are converted. To a first approximation, the conversion can be understood as turning the sequence of virtual bindings $[(l,k,n,\mathcal{E}[\![\mathsf{e}]\!]), (l,k',n',\mathcal{E}[\![\mathsf{e'}]\!]), \ldots]$ into a nested let-expression **let** n= e **in let** n'=e' **in** …. Incidentally, the code e' may refer to n, hence the order of virtual bindings is important: this accounts for the representation of virtual bindings as a sequence rather than a set. Virtual bindings with the same locus l and the memo key k belong to the same equivalence class, or group. The semantic operation groupby k is meant to group the bindings of the same locus. One let statement is generated per group, for the first binding in the group (the group representative). All other variables within the same group of virtual bindings are substituted with the group representative variable[3].

The interface for genlet described above differs from our previous proposal [Yallop and Kiselyov 2019] 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.

It turns out that the genletrec for generating (mutually) recursive definitions presented by Yallop and Kiselyov [2019] is a minor variant of the above genlet, with almost the same semantics. The presentation (and forthcoming full paper) will give details; for now, we refer the interested reader to the accompanying code.

It has been recognized early on [Bondorf 1992; Lawall and Danvy 1994] that one can use control effects (either direct or realized via CPS) to answer the compositionality challenge of the ordinary, well-nested let-insertion. Kameyama et al. [2011] give a comprehensive formal treatment. Unfortunately, neither 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. Generating (mutually) recursive bindings has not previously been formally considered at all, to our knowledge.

In summary, 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 Kameyama et al. [2011]; Kiselyov et al. [2016]; Yallop and Kiselyov [2019]).

Acknowledgements We are grateful to the anonymous reviewers for many helpful suggestions. We thank Yukiyoshi Kameyama for hospitality.

References

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. ISBN 0-89791-481-3. doi: 10.1145/141471.141483.

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: Staging with delimited control. *J. Funct. Program.*, 21(6):617–662, November 2011.

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.

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. ISBN 978-3-319-47957-6. doi: 10.1007/978-3-319-47958-3_15.

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, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182483.

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.

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. ISBN 978-1-4503-6226-9. doi: 10.1145/3294032.3294078.

A Base and Codec, formally

The base calculus, used for the generated code and the base of the generator, is the standard call-by-value simply-typed lambda-calculus with constants and ordinary let-expressions and (potentially mutually) recursive letrec-expressions. Figure 1 presents its syntax. There, c0, c1, c2 and c3 stand for constants of the corresponding arity. Applications of constants to fewer arguments than their arity are not considered expressions. The calculus includes integer and boolean literals (as zero-arity constants), the successor operation succ of arity 1 and binary arithmetic and comparison operations on integers, of obvious types.

The base calculus can be represented as OCaml signature. Calculus expressions of the type α are represented as OCaml values of the type α repr. The mutually recursive mletrec takes the collection of clauses indexed by idx; the first argument to mletrec tells the number of clauses.

[3] This substitution is cheap since $\mathcal{E}[e]$ is typically a map from an environment ρ to the appropriate V_t. The substitution of n' with n is $\lambda\rho.\mathcal{E}[e]\,\rho[n' \to n]$.

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

```
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 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 2. Base calculus represented in OCaml: its syntax as OCaml signature