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 combinators, is composition: 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 illScoped 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:

\[
\begin{align*}
t1 & := 1 + 2 \\
\text{sq} & := \lambda x. x \times x \\
gib & := \lambda x. \lambda y. \text{let rec } \text{loop } n = \\
& \quad \text{if } n=0 \text{ then } x \text{ else if } n=1 \text{ then } y \text{ else } \\
& \quad \text{loop } (n-1) + \text{ loop } (n-2) \text{ in } \text{loop } 5
\end{align*}
\]

The notation := is not part of the calculus; it is used to attach a name to an expression for easy reference. The function gib 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 Coded; each expression serves as a generator of the corresponding earlier Base expression:

\[
\begin{align*}
\text{ct1} & := \text{int } 1 + \text{ int } 2 \\
\text{csq} & := \lambda x. x \times x \\
gib5 & := \lambda x. \lambda y. \text{let rec } \text{loop } n = \\
& \quad \text{if } n=0 \text{ then } x \text{ else if } n=1 \text{ then } 1 \text{ else } \\
& \quad \text{loop } (n-1) + \text{ loop } (n-2) \text{ in } \text{loop } 5
\end{align*}
\]

Here int of the type \( \text{int} \to \text{int code} \) generates the code of an integer literal; \( \times \) of the type \( \text{int code} \to \text{int code} \to \text{int code} \) combines the code of summands to the code of the addition expression; \( \lambda x. \text{body} \) 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 \( \text{csq} \) and \( \text{gib5} \) represent \( \text{sq} \) (resp. \( \text{gib5} \)) 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: \( M[\cdot] \), for the meaning of whole programs, and \( E[\cdot] \), 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 \( \bot \)). \( M[\cdot] \) maps (a type derivation of a closed) expression \( e \) of type \( t \) to \( V_t \); \( E[\cdot|\cdot] \) maps \( t \) to a continuous function from the environment to \( V_t \). (We write only the expression rather than the entire type derivation, and often elide \( \Gamma \) and the type annotations to avoid clutter.) 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, which we will implicitly apply.

The environment \( \rho \) is a strict finite map from variable names to \( V_t \); the environment extension is written \( \rho[x \mapsto v] \). The environment’s domain also includes an extra, distinguished variable named \( \ell \), mapped to an element of \( V_{nom} \) – this is the wrinkle. The semantic
rules are entirely standard (modulo $\ell$). We show only the rules for abstraction and application:

\[
E_R[\lambda x.e] \rho = \lambda x. E_R[e] \rho[x:=x, \ell \mapsto 1; \rho(\ell)] \\
E_R[e_1 e_2] \rho = E_R[e_1] \rho[x:=x, \ell \mapsto 1; \rho(\ell)] (E_R[e_2] \rho[x:=x, \ell \mapsto 2; \rho(\ell)]) \\
M_R[\ell] = E_R[e] \{ [\ell \mapsto \emptyset] \}
\]

For now, $\ell$ is not actually used, and it might as well be absent. It will help later.

The second semantics of Base, notated by the superscript $S$, maps an expression to its symbolic form (a string, for example). Here every value 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 $E^C[-]$ of Codec is an extension of the R semantics of Base\(^1\). Since we what generate are (potentially open) think of generating function bodies Base expressions of type $t$, the meaning of a $t$ code value is $E^C[e]$: the meaning of $e$ according to $R$ or $S$ or some other semantics of Base\(^2\) We pair $E^C[e]$ of a sequence of so-called virtual bindings $v$ in the next section; they can be disregarded for now. The following are two sample sequences for the semantic function $E^C[-]$ of Codec, for generating an integer literal and an abstraction.

\[
E^C[\text{int } i] = E^C[i], \emptyset \quad E^C[\text{int } x] = \text{mk} (E^C[e] \rho[x:=\langle E^C[n], \emptyset \rangle, \ell \mapsto 1; \rho(\ell)])
\]

where $n = \rho(\ell)$

\[
\text{mk} (E^C[e], v) = E^C[\lambda n.c], v
\]

When we generate an abstraction, the current association of $\ell$ in the environment acts as the name for the bound variable. It is easy to see that $\rho(\ell)$ truly gives us a fresh name.

If we use the R semantics for the generated code (that is, choose $E^C[-]$ to be $E_R[-]$) we see that $M^C_R[\text{et}]$ is exactly $M_R[\text{tl}]$ (which is the integer 3), $M^C_S[\text{et}]$ and $M^C_S[\text{tl}]$ both mean the squaring function, and $M^C_R[\text{eq}]$ and $M^C_R[\text{gt}]$ both mean the function that takes two arguments $x$ and $y$ and returns the sum of 5 copies of $x$ and 3 copies of $y$.

If we use the S semantics (with $\text{et}$ and $\text{tl}$) still coincide (both mean the string 1+2). $M^C_S[\text{et}]$ and $M^C_S[\text{tl}]$ are generally different but alpha-equivalent lambda-expressions strings. $M^C_R[\text{eq}]$ is the new string

\[
\lambda x.\lambda y. ((y + x) + y) + (y + x) + ((y + x) + y)
\]

It is an 'optimized' version of $\text{gb5}$, 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 $\text{genlet } l_e m e$, $l$ is a locus variable (previously bound by let locus), $m$ 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 $\text{cgib5}$:

\[
\text{cgib5} := \lambda x.\lambda y. \text{let locus } l \in \text{let rec } \ell n = 0 \text{ if } n=0 \text{ then } x \text{ else if } n=1 \text{ then } 1 \text{ else } \text{genlet } l ((n-1) \text{ (loop (n-1)) } + \text{genlet } l ((n-1) \text{ (loop (n-2))} \text{ in } \ell 5
\]

Using the semantics of these operations, explained above, we can see that whereas $M^C_R[\text{cgib5}]$ remains the same as $E_R[\text{gib5}]$, $M^C_S[\text{cgib5}]$ is the string

\[
\lambda x.\lambda y. \text{let } z = y \text{ in } u = x \text{ in } \text{let } v = z + u \text{ in } \text{let } w = v + z \text{ in } \text{let } x = w + v \text{ in } x w + w
\]

which is indeed an optimized version of $\text{gib5}$, without either loops or duplication.

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

The semantic rules are as follows

\[
E^C[\text{genlet } l_e m e] = E^C[n], \nu(l, k, n, e_k)
\]

where $l = \rho(\ell)$

\[
k = E^C[\text{m}] \rho[\ell \mapsto 1; \rho(\ell)]
\]

\[
n = \rho(\ell)
\]

\[
(e_k, v) = E^C[e] \rho[\ell \mapsto 2; \rho(\ell)]
\]

\[
E^C[\text{let locus } l \text{ in } e] = ((bg \ g1 \ldots (bg \ gn \ E^C[e])); v')
\]

where $l = \rho(\ell)$

\[
\nu' = ((l', k, n, e_k) | (l', k, n, e_k) \nu \nu', l' \parallel \ell )
\]

\[
[g1 \ldots gn] = \text{groupby } k \{(l', k, n, e_k) | (l', k, n, e_k) \nu = v \}
\]

\[
bg ((l, k, n, E^C[e]), \ldots, n1, \ldots) = E^C[e] = \text{let } E^C[c] = \text{subst } [n1 \rightarrow n, \ldots] E^C[e] \text{ in } E^C[\text{let } n = e \text{ in } c]
\]

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 $l \text{ in } e$ 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 $[(l, k, n, E^C[e]), (l', k, n', E^C[e]), \ldots]$ into a nested let-expression

\[
\text{let } n = e \text{ in } \text{let } n' = e' \text{ in } \ldots
\]

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

\[\text{For a two-level language, the S semantics for Codec in unnecessary.}\]

\[\text{The absence of the subscript in } E \text{ means we are talking about any semantics.}\]
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 memoziation 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 \(\lambda y \ldots\), 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 genlet 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 thank Yukiyoshi Kameyama for hospitality.

**References**


Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers for generating (mutually) recursive \(\lambda\rho\) operations on integers, of obvious types.

\[ \begin{array}{l}
\text{Variables} \quad x, y, z, u, f, n, r, \ldots \\
\text{Types} \quad t ::= \text{int} | \text{bool} | t \to t \\
\text{Expressions} \quad e ::= x | c0 | c1 e | c2 e e | c3 e e e | \lambda x. e | e e \\
\text{Values} \quad v ::= c0 | \lambda x. e
\end{array} \]

**Figure 1.** Syntax of the base calculus; \(c_i\) are constants of arity \(i\).

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

\[ \lambda \sigma \cdot \text{letrec} \{ \text{let } \ldots \} \text{ in } e \]

\[ \text{let } (\text{rec}) \text{ insertion without effects, lights or magic} \]
type $\alpha$ repr

val lam : ($\alpha$ repr $\rightarrow$ $\beta$ repr) $\rightarrow$ ($\alpha$ $\rightarrow$ $\beta$) repr
val let_ : $\alpha$ repr $\rightarrow$ ($\alpha$ repr $\rightarrow$ $\beta$ repr) $\rightarrow$ $\beta$ repr
val (/) : ($\alpha$ $\rightarrow$ $\beta$) repr $\rightarrow$ ($\alpha$ repr $\rightarrow$ $\beta$ repr) (* application *)
val if_ : bool repr $\rightarrow$ $\alpha$ repr $\rightarrow$ $\alpha$ repr $\rightarrow$ $\alpha$ repr

val int : int $\rightarrow$ int repr
val bool : bool $\rightarrow$ bool repr
val succ : int repr $\rightarrow$ int repr
val (+) : int repr $\rightarrow$ int repr $\rightarrow$ int repr
val (−) : int repr $\rightarrow$ int repr $\rightarrow$ int repr
val (*) : int repr $\rightarrow$ int repr $\rightarrow$ int repr
val (=.) : int repr $\rightarrow$ int repr $\rightarrow$ bool repr

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