let (rec) insertion without Effects, Lights or Magic

Oleg Kiselyov Jeremy Yallop

Tohoku University, Japan

University of Cambridge, UK

PEPM2020
January 18, 2022

arXiv:2201.00495

Outline

Summary

- What let-insertion actually *means*
- The first formal model that uniformly treats let-insertion, letrec- insertion and mutually letrec-insertion
- *No* continuation or state effects
- Not just theory:
  - Executable semantics: the way to implement let(rec) insertion in *any code generation framework*, without any coroutines, delimited continuations or other run-time or compiler magic
  - Simpler than before interface for (mutual) letrec insertion
  - Implemented in the current MetaOCaml

Code Generation: Code Template

```
printf "(%s + %d) * %s" e1 n e2
```

```
`(* (+ ,e1 ,n) ,e2)
```

Code Combinators

$$(\texttt{e1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{n})\ \underline{\texttt{*}}\ \texttt{e2}$$

where

$$\begin{aligned} \texttt{e1,e2} &: \texttt{int code} \\ \underline{\texttt{+}},\underline{\texttt{*}} &: \texttt{int code} \rightarrow \texttt{int code} \rightarrow \texttt{int code} \\ \underline{\texttt{int}} &: \texttt{int} \rightarrow \texttt{int code} \end{aligned}$$

Code Combinators

$$\texttt{(e1 } \underline{\texttt{+}} \ \underline{\texttt{int}} \texttt{ n) } \underline{\texttt{*}} \texttt{ e2}$$

$$\underline{\lambda}(\lambda \texttt{x. (}\underline{\texttt{int}}\texttt{ 1 } \underline{\texttt{+}} \ \underline{\texttt{int}} \texttt{ 2) } \underline{\texttt{+}} \texttt{ x)}$$

where

$$\begin{aligned}
\texttt{e1,e2} &: \texttt{int code} \\
\underline{\texttt{+}},\underline{\texttt{*}} &: \texttt{int code} \rightarrow \texttt{int code} \rightarrow \texttt{int code} \\
\underline{\texttt{int}} &: \texttt{int} \rightarrow \texttt{int code} \\
\underline{\lambda} &: (\alpha\,\texttt{code} \rightarrow \beta\,\texttt{code}) \rightarrow (\alpha \rightarrow \beta)\texttt{code}
\end{aligned}$$

Code Combinators

$$(\texttt{e1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{n})\ \underline{\texttt{*}}\ \texttt{e2}$$

$$\underline{\lambda}\texttt{x.}\ (\underline{\texttt{int}}\ \texttt{1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{2})\ \underline{\texttt{+}}\ \texttt{x}$$

where

$$\begin{aligned} \texttt{e1,e2} &: \texttt{int code} \\ \underline{\texttt{+}},\underline{\texttt{*}} &: \texttt{int code} \rightarrow \texttt{int code} \rightarrow \texttt{int code} \\ \underline{\texttt{int}} &: \texttt{int} \rightarrow \texttt{int code} \\ \underline{\lambda} &: (\alpha\,\texttt{code} \rightarrow \beta\,\texttt{code}) \rightarrow (\alpha \rightarrow \beta)\texttt{code} \end{aligned}$$

Code Combinators

$$\texttt{(e1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{n)}\ \underline{\texttt{*}}\ \texttt{e2}$$

$$\underline{\lambda}\texttt{x.}\ \texttt{(}\underline{\texttt{int}}\ \texttt{1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{2)}\ \underline{\texttt{+}}\ \texttt{x}$$

$$\rightsquigarrow\ \texttt{"fun x7 -> (1+2) + x7"}$$

where

$$\begin{aligned}
\texttt{e1,e2} &: \texttt{int code} \\
\underline{\texttt{+}},\underline{\texttt{*}} &: \texttt{int code} \to \texttt{int code} \to \texttt{int code} \\
\underline{\texttt{int}} &: \texttt{int} \to \texttt{int code} \\
\underline{\lambda} &: (\alpha\,\texttt{code} \to \beta\,\texttt{code}) \to (\alpha \to \beta)\texttt{code}
\end{aligned}$$

Code Combinators

$$(\texttt{e1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{n})\ \underline{\texttt{*}}\ \texttt{e2}$$

$$\underline{\lambda}\texttt{x.}\,(\underline{\texttt{int}}\ \texttt{1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{2})\ \underline{\texttt{+}}\ \texttt{x}$$
$$\leadsto \texttt{"fun x7 -> (1+2) + x7"}$$

$$\underline{\lambda}\texttt{x.}\,\underline{\texttt{let}}\ (\underline{\texttt{int}}\ \texttt{1}\ \underline{\texttt{+}}\ \underline{\texttt{int}}\ \texttt{2})\ \lambda\texttt{y.}\,\texttt{y}\ \underline{\texttt{+}}\ \texttt{x}$$
$$\leadsto \texttt{"fun x7 -> let y8 = (1+2) in y8 + x7"}$$

where

$$\begin{aligned}
\texttt{e1,e2} &: \texttt{int code} \\
\underline{\texttt{+}},\underline{\texttt{*}} &: \texttt{int code} \to \texttt{int code} \to \texttt{int code} \\
\underline{\texttt{int}} &: \texttt{int} \to \texttt{int code} \\
\underline{\lambda} &: (\alpha\,\texttt{code} \to \beta\,\texttt{code}) \to (\alpha \to \beta)\texttt{code} \\
\underline{\texttt{let}} &: \alpha\,\texttt{code} \to (\alpha \to \beta)\texttt{code} \to \beta\,\texttt{code}
\end{aligned}$$

Outline

Compositionality. . . and the lack of it

$$(\underline{\lambda}x.\ \underline{\mathtt{let}}\ (\underline{\mathtt{int}}\ 1\ \underline{+}\ \underline{\mathtt{int}}\ 2)\ (\lambda y.\ y\ \underline{+}\ x))$$

$$\leadsto\ \texttt{"(fun x7 -> let y8 = (1+2) in (y8 + x7))"}$$

Compositionality. . . and the lack of it

$$(\underline{\lambda}\texttt{x}.\ \underline{\texttt{let}}\ (\underline{\texttt{int}}\ 1\ \underline{+}\ \underline{\texttt{int}}\ 2)\ (\lambda\texttt{y}.\ \texttt{y}\ \underline{+}\ \texttt{x}))$$

$$\rightsquigarrow \texttt{"(fun x7 -> let y8 = (1+2) in (y8 + x7))"}$$

let-insertion

$$(\underline{\lambda}\texttt{x}.\ (\underline{\texttt{glet}}\ (\underline{\texttt{int}}\ 1\ \underline{+}\ \underline{\texttt{int}}\ 2)\ \underline{+}\ \texttt{x}))$$

$$\rightsquigarrow \texttt{"let y8 = (1+2) in (fun x7 -> (y8 + x7))"}$$

where

$$\underline{\text{glet}} : \alpha\ \text{code} \to \alpha\ \text{code}$$

Sharing

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

⇝ `"(((6 + 7) + 20) * ((6 + 7) + 30)) / 100"`

Sharing

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

⇝ `"(((6 + 7) + 20) * ((6 + 7) + 30)) / 100"`

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

⇝
```
"let x4 = (6 + 7) in
 let x5 = x4 + 20 in
 let x6 = x4 + 30 in
 (x5 * x6) / 100"
```

Outline

Introduction

let-insertion

▶ **Definitions**

Parameterized, recursive definitions

Conclusions

Definitions

> *"...the definitions are not part of our subject, but are, strictly speaking, mere typographical conveniences....*
>
> *In spite of the fact that definitions are theoretically superfluous, it is nevertheless true that they often convey more important information than is contained in the propositions in which they are used. ...*
>
> *The collection of definitions embodies our choice of subjects and our judgement as to what is most important. Secondly, ...the definition contains an analysis of a common idea, and may therefore express a notable advance."*

Whitehead & Russell. Principia mathematica, volume I. Cambridge Univ. Press, 1910, p12

Definitions

> *"... the definitions are not part of our subject, but are, strictly speaking, mere typographical conveniences....*
>
> *In spite of the fact that definitions are theoretically superfluous, it is nevertheless true that they often convey more important information than is contained in the propositions in which they are used. ...*
>
> *The collection of definitions embodies our choice of subjects and our judgement as to what is most important. Secondly, ... the definition contains an analysis of a common idea, and may therefore express a notable advance."*

Whitehead & Russell. Principia mathematica, volume I. Cambridge Univ. Press, 1910, p12

Definitions

> *"...the definitions are not part of our subject, but are, strictly speaking, mere typographical conveniences....*
>
> *In spite of the fact that definitions are theoretically superfluous, it is nevertheless true that they often convey more important information than is contained in the propositions in which they are used. ...*
>
> *The collection of definitions embodies our choice of subjects and our judgement as to what is most important. Secondly, ...the definition contains an analysis of a common idea, and may therefore express a notable advance."*

Whitehead & Russell. Principia mathematica, volume I. Cambridge Univ. Press, 1910, p12

The main difficulty of making definitions

- Definitions precede uses
  in the finished text
- In writing, (attempted) use precedes definition

The main difficulty of making definitions

- Definitions precede uses
  in the finished text
- In writing, (attempted) use precedes definition

Definitions are made in hindsight

They are read forwards, but generated backwards

An example of making a definition

```
\begin{frame}{Sharing}

\begin{tabular}[C]{l}
let x = \textcolor{red}{(}_
```

An example of making a definition

```
\documentclass{beamer}
\newcommand{\lam}{\quant\lambda}

\title{\textsf{let} (\textsf{rec}) insertion
without\\Effects, Lights or Magic}
```

Going back

An example of making a definition

```
\documentclass{beamer}
\newcommand{\lam}{\quant\lambda}

\def\rbra{\textcolor{red}(}

\title{\textsf{let} (\textsf{rec}) insertion
without\\Effects, Lights or Magic}
```

Making a definition

An example of making a definition

```
\begin{frame}{Sharing}

\begin{tabular}[C]{l}
let x = \rbra_
```

Returning (resuming)

A different approach

Margin notes

Outline

Introduction

let-insertion

Definitions

- **Parameterized, recursive definitions**

Conclusions

Ackermann function

```
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
```

Ackermann function generator

```
letrec λack.λm.λn.
  if (m = int 0) (n + int 1)
  (if (n = int 0) (ack @ (m - int 1) @ (int 1))
    (ack @ (m - int 1) @ (ack @ m @ (n - int 1))))
(λack.  ack @ int 2)
```

Specialized Ackermann function generator

```
let rec ack = λm.λn.
   if m=0 then n + (int 1) else
   if (n = int 0)
      (gletrec (m-1) (ack (m-1)) @ int 1)
      (gletrec (m-1) (ack (m-1)) @
                           (gletrec m (ack m) @ (n-int 1)))
in gletrec 2 (ack 2)
```

Specialized Ackermann function generator

```
let rec ack = λm.λn.
   if m=0 then n + (int 1) else
   if (n = int 0)
      (gletrec (m-1) (ack (m-1)) @ int 1)
      (gletrec (m-1) (ack (m-1)) @
                                  (gletrec m (ack m) @ (n-int 1)))
in gletrec 2 (ack 2)
```

```
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
```

Sharing, again

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

$\leadsto$

```
"let x4 = (6 + 7) in
 let x5 = x4 + 20 in
 let x6 = x4 + 30 in
 (x5 * x6) / 100"
```

Outline

Conclusions

- What let-insertion actually *means*
- The first formal model that uniformly treats let-insertion, letrec- insertion and mutually letrec-insertion
- *No* continuation or state effects
- Not just theory:
  - Executable semantics: the way to implement let(rec) insertion in *any code generation framework*, without any coroutines, delimited continuations or other run-time or compiler magic
  - Simpler than before interface for (mutual) letrec insertion
  - Implemented in the current MetaOCaml