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

▶ Introduction

let-insertion

Definitions

Parameterized, recursive definitions

Conclusions
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

\[(e_1 + \text{int } n) \ast e_2\]

where

\[e_1, e_2 : \text{int code}\]
\[+,* : \text{int code} \rightarrow \text{int code} \rightarrow \text{int code}\]
\[\text{int} : \text{int} \rightarrow \text{int code}\]
Code Combinators

\[(e_1 + \text{int } n) \ast e_2\]

\[\lambda(\lambda x. (\text{int } 1 + \text{int } 2) + x)\]

where

\[e_1, e_2 : \text{int code}\]

\[\ast, + : \text{int code} \rightarrow \text{int code} \rightarrow \text{int code}\]

\[\text{int} : \text{int} \rightarrow \text{int code}\]

\[\lambda : (\alpha \text{ code} \rightarrow \beta \text{ code}) \rightarrow (\alpha \rightarrow \beta)\text{code}\]
Code Combinators

\[(e_1 + \text{int } n) \ast e_2\]

\[\lambda x. (\text{int } 1 + \text{int } 2) + x\]

where

\[e_1, e_2 : \text{int code}\]

\[\ast, + : \text{int code} \rightarrow \text{int code} \rightarrow \text{int code}\]

\[\text{int} : \text{int} \rightarrow \text{int code}\]

\[\lambda : (\alpha \text{ code} \rightarrow \beta \text{ code}) \rightarrow (\alpha \rightarrow \beta)\text{code}\]
Code Combinators

\[(e_1 + \text{int } n) \ast e_2\]

\[\lambda x. (\text{int } 1 + \text{int } 2) + x\]

\[\rightsquigarrow "\text{fun } x7 \rightarrow (1+2) + x7"\]

where

\[e_1, e_2 : \text{int code}\]

\[\ast, + : \text{int code} \rightarrow \text{int code} \rightarrow \text{int code}\]

\[\text{int} : \text{int} \rightarrow \text{int code}\]

\[\lambda : (\alpha \text{ code} \rightarrow \beta \text{ code}) \rightarrow (\alpha \rightarrow \beta)\text{code}\]
Code Combinators

\[(e1 + \text{int } n) \ast e2\]

\[\lambda x. (\text{int } 1 + \text{int } 2) + x\]

\[\leadsto \text{"fun } x7 \rightarrow (1+2) + x7\"

\[\lambda x. \text{let } (\text{int } 1 + \text{int } 2) \lambda y. y + x\]

\[\leadsto \text{"fun } x7 \rightarrow \text{let } y8 = (1+2) \text{ in } y8 + x7\"

where

\[e1, e2 : \text{int code}\]

\[\_, +, \ast : \text{int code} \rightarrow \text{int code} \rightarrow \text{int code}\]

\[\text{int} : \text{int} \rightarrow \text{int code}\]

\[\lambda : (\alpha \text{ code} \rightarrow \beta \text{ code}) \rightarrow (\alpha \rightarrow \beta)\text{code}\]

\[\text{let} : \alpha \text{ code} \rightarrow (\alpha \rightarrow \beta)\text{code} \rightarrow \beta \text{ code}\]
Outline

Introduction

▶ let-insertion

Definitions

Parameterized, recursive definitions

Conclusions
Compositionality...and the lack of it

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

$\leadsto "(\text{fun } x7 \rightarrow \text{let } y8 = (1+2) \text{ in } (y8 + x7))"$
Compositionality... and the lack of it

\[(\lambda x. \text{let } (\text{int } 1 + \text{int } 2) \, (\lambda y. y + x))\]
\[\mapsto "(\text{fun } x7 \rightarrow \text{let } y8 = (1+2) \, \text{in } (y8 + x7))"\]

let-insertion

\[(\lambda x. \text{glet } (\text{int } 1 + \text{int } 2) + \, x)\]
\[\mapsto "\text{let } y8 = (1+2) \, \text{in } (\text{fun } x7 \rightarrow (y8 + x7))"\]

where

\[\text{glet : } \alpha \text{ code } \rightarrow \alpha \text{ code}\]
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 \textsf{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 = \textbackslash rbra\_
\end{tabular}

Returning (resuming)
A different approach

Margin notes
Outline

Introduction

let-insertion

Definitions

▶ Parameterized, recursive definitions

Conclusions
Ackermann function

let rec ack = \m.\n.  
  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 = \m. \n. 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 = \m. \n. 
    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 = \u. if u = 0 then y 1 else y (x (u - 1))
and y = \v. if v = 0 then z 1 else z (y (v - 1))
and z = \w. w + 1
in x
let x = \texttt{glet (int 6 + int 7)} in
(\texttt{glet (x + int 20)} * \texttt{glet (x + int 30)}) / \texttt{int 100}

\texttt{"let x4 = (6 + 7) in}
let x5 = x4 + 20 in
\texttt{let x6 = x4 + 30 in}
(x5 * x6) / 100"
Outline

Introduction

let-insertion

Definitions

Parameterized, recursive definitions

► Conclusions
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