Understanding the Expressive Power of Unhygienic Substitution in Metaprogramming via Combinatory Logic

Martin Lester

Department of Computer Science,
University of Reading

Computational Logic and Applications, 2019–07–01

Combinatory Logic

- Combinatory logic is a system of computation using term rewriting.
- With just two basic combinators, S and K, and their associated rewrites, we achieve Turing completeness:
- $S\ f\ g\ x \rightarrow f\ x\ (g\ x)$
- $K\ x\ y \rightarrow x$
- Other base combinators are often used for efficiency or convenience:
- $I\ x \rightarrow x$
- $B\ f\ g\ x \rightarrow f\ (g\ x)$
- $C\ f\ g\ x \rightarrow f\ x\ g$

Bracket Abstraction

Translate a lambda abstraction into an extensionally equivalent term of combinatory logic:

- $A_x[e_1\, e_2] \mapsto S\ (A_x[e_1])\ (A_x[e_2])$
- $A_x[x] \mapsto I$
- $A_x[c] \mapsto K\ c$ — where $c \neq x$

Naive translation can produce large terms.

Why Combinatory Logic?

- Combinatory logic is one of the simplest computational systems that is Turing-powerful.
- It avoids substitution, which is the fiddliest part of lambda calculus.
- Its simplicity makes it a candidate for hardware or virtual machines for executing functional programs.
- Is it expressive?
- Clearly, it can express any computation of a lambda calculus expression or Turing machine.
- But can it do so efficiently and succinctly?

Metaprogramming

- Programs that transform other programs are a form of metaprogramming.
- Extensional: Lisp, Template Haskell, MetaOCaml.
- Intensional: ReFLect, SF-calculus.
- Unstructured: JavaScript eval.
- Idea: Can we understand power of metaprogramming by translation to other formalisms?

Outline

Metaprogramming in $\lambda_e^{\to\Box}$

- Davies and Pfenning's $\lambda_e^{\to\Box}$ is one of the most influential metaprogramming systems.
- Idea: Extend λ-calculus with 2 constructs:
  - $box\ E$ — treat expression E as a code value
  - $let\ box\ u = E_1\ in\ E_2$ — unwrap code value E_1; bind to u in E_2
- $let\ box\ \ldots$ can splice code values together and run them.
- Evaluation: $let\ box\, u = box\, E_1\ in\ E_2 \mapsto [E_1/u]E_2$
- No evaluation under a box.

Metaprogramming in $\lambda_e^{\to\Box}$

Splicing of code values:

$$\begin{array}{l} \mathit{let}\ \ \mathit{box}\ \ u = x\ \mathit{in} \\ \mathit{let}\ \ \mathit{box}\ \ v = y\ \mathit{in} \\ \mathit{box}\ (u\ v) \end{array}$$

Eval function:

$$\lambda x.\ \mathit{let}\ \ \mathit{box}\ \ u = x\ \mathit{in}\ \ u$$

Type System for $\lambda_e^{\to\Box}$

$$\frac{x{:}A \text{ in } \Gamma}{\Delta; \Gamma \vdash^e x : A}\ \text{ovar}$$

$$\frac{\Delta; (\Gamma, x{:}A) \vdash^e E : B}{\Delta; \Gamma \vdash^e \lambda x{:}A.\ E : A \to B} \to \text{I}$$

$$\frac{\Delta; \Gamma \vdash^e E_1 : B \to A \qquad \Delta; \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e E_1\ E_2 : A} \to \text{E}$$

- $\lambda_e^{\to\Box}$ type system ensures code with free variables can't be run.
- Typing rules extend simply-typed λ-calculus; similar to modal logic S4.
- Type $\Box A$ means code that evaluates to something of type A.
- Second context Δ is used to track variables with modal types.

Type System for $\lambda_e^{\to\Box}$

$$\frac{u{:}A \text{ in } \Delta}{\Delta; \Gamma \vdash^e u : A}\ \text{mvar}$$

$$\frac{\Delta; \cdot \vdash^e E : A}{\Delta; \Gamma \vdash^e \mathbf{box}\ E : \Box A}\ \Box\text{I}$$

$$\frac{\Delta; \Gamma \vdash^e E_1 : \Box A \qquad (\Delta, u{:}A); \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 : B}\ \Box\text{E}$$

- $\lambda_e^{\to\Box}$ type system ensures code with free variables can't be run.
- Typing rules extend simply-typed λ-calculus; similar to modal logic S4.
- Type $\Box A$ means code that evaluates to something of type A.
- Second context Δ is used to track variables with modal types.

Semantics of Metaprogramming

Operational semantics of metaprogramming systems can be intricate, but here are straightforward.

Attempts to produce a denotational semantics for metaprogramming systems struggle with the following problem:

What is the meaning of open code?

Kiselyov's Combinatory Translation

- Kiselyov recently proposed a new translation from λ-calculus to combinatory logic.
- Assumes λ-terms encoded using de Bruijn indices.
- Practical application: Efficient (time and space) implementation of functional programming languages.
- By product: Denotation of λ-terms using combinatory logic, including open terms.
  - Closed terms — obviously, same function as a combinator term.
  - Open terms — function when closed with sufficient λs.

Kiselyov's Combinatory Translation

$\Gamma \vdash e : \tau$	$\mathcal{E}\,[\Gamma \vdash e : \tau]$

$$\frac{}{\tau \vdash z : \tau}\ Var \qquad \frac{}{\tau \models I}\ EVar$$

$$\frac{\Gamma^{+} \vdash e : \tau}{\Gamma^{+}, \sigma \vdash s\ e : \tau}\ W \qquad \frac{\Gamma^{+} \models d}{\Gamma^{+}, \sigma \models (\models K)\ \amalg\ (\Gamma^{+} \models d)}\ EW$$

$$\frac{\vdash e : \tau}{\vdash \lambda\, e : \sigma \to \tau}\ Abs_0 \qquad \frac{\models d}{\models K\, d}\ EAbs_0$$

$$\frac{\Gamma, \sigma \vdash e : \tau}{\Gamma \vdash \lambda\, e : \sigma \to \tau}\ Abs \qquad \frac{\Gamma, \sigma \models d}{\Gamma \models d}\ EAbs$$

$$\frac{\Gamma_1 \vdash e_1 : \sigma \to \tau \quad \Gamma_2 \vdash e_2 : \sigma}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1\ e_2 : \tau}\ App \qquad \frac{\Gamma_1 \models d_1 \quad \Gamma_2 \models d_2}{\Gamma_1 \sqcup \Gamma_2 \models (\Gamma_1 \models d_1)\ \amalg\ (\Gamma_2 \models d_2)}\ EApp$$

- Type environments ordered to match order of binding.
- Weakening only as explicitly allowed.

Extending Kiselyov's Translation to $\lambda_e^{\to\Box}$

- Davies and Pfenning propose this translation from $\lambda_e^{\to\Box}$ into λ-calculus:
- $\Box A \mapsto \mathit{unit} \to A$
- $\mathit{box}\ \ E \mapsto \lambda x : \mathit{unit}\,.E$
- $\mathit{let}\ \ \mathit{box}\, u = E_1\, \mathit{in}\, E_2 \mapsto \lambda x : \mathit{unit} \to A.[x()/u]E_2)E_1$
- Effectively, code values becomes thunks.

Extending Kiselyov's Translation to $\lambda_e^{\to\Box}$

- Idea: Apply this translation, then Kiselyov's translation to combinatory logic.
- Problem 1: Kiselyov's translation assumes de Bruijn indices.
  - OK: *let box* acts as a binder, observing scoping.
- Problem 2: Type system has 2 contexts.
  - OK: Typeability preserved for system with single context.
- Can create a new type system with new translation rules.

Unhygienic Substitution and $\lambda_{\mathcal{S}}$

- $\lambda_e^{\rightarrow\Box}$ lacks one useful feature: unhygienic substitution.
- $\lambda_{\mathcal{S}}$ adds this and an *unbox* primitive for more convenient splicing.
- Evaluation: $\mathit{unbox}\,(\mathit{box}\,v) \rightarrow v$
- Example:

$$
\begin{array}{l}
\mathit{let}\ \ a = \mathit{box}\ \ x\ \mathit{in} \\
\mathit{let}\ \ b = \mathit{box}\,(\lambda x.\lambda y.(\mathit{unbox}\,a) + y)\ \mathit{in} \\
(\mathit{run}\ \ b)\,1\,1
\end{array}
$$

Extending Kiselyov's Translation to $\lambda_{\mathcal{S}}$

- Choi and others propose an unstaging translation to remove metaprogramming from $\lambda_{\mathcal{S}}$.
- Can we apply this, as before?
- Translation is based around passing records to simulate environments.
- Effectively pushes comparison of variable names to run-time.
- Not meaningful, if goal is to understand power of variables.
- If undiscerning, can find a linear translation of anything: interpreter/translator + original program.

Extending Kiselyov's Translation to λ_S

- Problem: No longer have consistent binding order of variables.
- Translation to de Bruijn indices no longer works.
- $\lambda_e^{\rightarrow\Box}$ basically avoided problem of open code by forbidding unhygienic substitution.

The Problem of Open Code

Recall:

$$\begin{array}{l} \mathit{let}\ \ a = \mathit{box}\ \ x\ \mathit{in} \\ \mathit{let}\ \ b = \mathit{box}\ (\lambda x.\lambda y.(\mathit{unbox}\ a) + y)\ \mathit{in} \\ (\mathit{run}\ \ b)\ 11 \end{array}$$

What about:

$$\begin{array}{l} \mathit{let}\ \ a = \mathit{box}\ \ y\ \mathit{in} \\ \mathit{let}\ \ b = \mathit{box}\ (\lambda x.\lambda y.(\mathit{unbox}\ a) + y)\ \mathit{in} \\ (\mathit{run}\ \ b)\ 11 \end{array}$$

Or even:

$$\begin{array}{l} \mathit{let}\ \ a = \mathit{if}\ (*)\ \ \mathit{then}\ \ \mathit{box}\ \ x\ \mathit{else}\ \ \mathit{box}\ \ y\ \mathit{in} \\ \mathit{let}\ \ b = \mathit{box}\ (\lambda x.\lambda y.(\mathit{unbox}\ a) + y)\ \mathit{in} \\ (\mathit{run}\ \ b)\ 11 \end{array}$$

The Problem of Open Code

Consider:

$$
\begin{array}{l}
\mathit{let}\ \ a = \mathit{box}\ \ f_1(f_2(f_3(f_4 0)))\ \mathit{in} \\
\mathit{let}\ \ b = \mathit{box}\ (\lambda f_{i_1}.\lambda f_{i_2}.\lambda f_{i_3}.\lambda f_{i_4}.\ \mathit{unbox}\ a)\ \mathit{in} \\
(\mathit{run}\ \ b)\ g_1\ g_2\ g_3\ g_4
\end{array}
$$

Keeping the idea of combinatory logic terms as denotation, for n functions, the denotation of a would need to become a set of $n!$ different terms.

Conclusion

Contributions:

- Extension of compositional translation from λ-calculus to combinatory logic:
  - Successful for $\lambda_e^{\rightarrow\Box}$.
  - Unsuccessful for $\lambda_{\mathcal{S}}$.
- Argument that failure of extension for $\lambda_{\mathcal{S}}$ demonstrates expressive power of unhygienic (capturing) substitution.

Future work:

- Implement a translation of an OCaml-like language with metaprogramming to combinatory calculus.
- What is the story for intensional metaprogramming and SF-calculus?

Conclusion

Contributions:

- Extension of compositional translation from λ-calculus to combinatory logic:
  - Successful for $\lambda_e^{\rightarrow\Box}$.
  - Unsuccessful for $\lambda_{\mathcal{S}}$.
- Argument that failure of extension for $\lambda_{\mathcal{S}}$ demonstrates expressive power of unhygienic (capturing) substitution.

Future work:

- Implement a translation of an OCaml-like language with metaprogramming to combinatory calculus.
- What is the story for intensional metaprogramming and SF-calculus?

Thanks for listening. Questions are welcome.