Randomly-Scoped Lambda Calculus

Ben Blum

bblum@cs.cmu.edu

Abstract

Gaze upon the following travesty, which Python hath wrought upon the world:

```
>>> x
NameError: name 'x' is not defined
>>> [x for x in 1,2,3]
[1, 2, 3]
>>> x
3
>>>
```

The question of *scope* pertains to the set of rules governing which values a program's variables refer to during evaluation. Prior work has proposed two main approaches: *static* (or *lexical*) *scope*, in which variable bindings are resolved through source code analysis, and *dynamic scope*, in which bindings are resolved at run-time using a stack of activation records.

In this work, I present *random scope*, a new technique for scoping, as an alternative to the above two. Random scope affords programmers the flexibility to refer to multiple different binding sites simultaneously; for example, applying the term $(\lambda x.(\lambda x.x))$ to two arguments could evaluate to either the first or the second one. I figure out what kind of evaluation semantics would make this work, and then pretty much stumble onwards from there trying to make sense of the whole damn thing.

I define the Randomly-Scoped Lambda Calculus, and show how it can be used to compute random natural numbers, while also providing more popular deterministic functionalities such as factorial and fibonacci.

Keywords static scope, dynamic scope, lower is better

1. Introduction

Some modern programming languages [707, BR99, Chu85, DCH03, Goo12, Hoa12, Ha11, Pey03, SV12] include a mechanism to refer to previously-computed values using more consise names, typically called *variables*. The question then arises: "Which values should my variables refer to?" In a half-hearted attempt to answer, these languages implement *scope* (which I explained in the abstract; go read it), and hope it all works out okay.

Whether resolved statically or dynamically, modern scoping approaches lack the flexibility to refer to multiple binding sites simultaneously. I present the Randomly-Scoped Lambda Calculus (RSLC), in which references to shadowed variables can evaluate to any of their binding sites with equal probability.

Consider the following lambda calculus term:

$$(\lambda x.(\lambda x.x))\ A\ B$$

In ordinary lambda calculus, this always evaluates to B (quite unforgivingly so, in my opinion). However, I give the programmer the benefit of the doubt, and assume they named the first argument x as well because they wanted it to have equal opportunity. Hence, in RSLC this term can evaluate to either A or B. (On the other side of the coin [Blu12], RSLC encourages good programming practice: a programmer who *doesn't* want x to evaluate to A should name the first argument something different, to make their intentions more clear.)

In this paper I make the following contributions:

1. The **Randomly-Scoped Lambda Calculus (RSLC)**, a logical programming language that uses a novel syntactic technique called **random scoping**,
2. A new schema for general recursion in RSLC (because the Y combinator doesn't work anymore),
3. Programming examples in RSLC that employ random scope to compute random natural numbers,
4. Programming examples in RSLC that avoid random scope to implement deterministic arithmetic.

2. Language Definition

RSLC is an extention to the lazy un(i)typed lambda calculus with slightly modified substitution rules. In the grammar (Figure 1), a variable x carries around a list of expressions $\overline{e}$ which we have "tried to substitute for it before". When programming in RSLC, we simply write x (or y or

SIGBOVIK '13 Pittsburgh, PA, USA

$$
\begin{array}{rcll}
e & = & \lambda x.e \mid e_1 e_2 & \textit{functions} \\
& \mid & x[\overline{e}] & \textit{random variables} \\
& \mid & \mathsf{Z} \mid \mathsf{S}(e) & \textit{natural numbers} \\
& \mid & \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(x) \Rightarrow e_3 & \textit{natural induction} \\
& \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \textit{eager evaluation}
\end{array}
$$

Figure 1. RSLC formal grammar.

$e \mapsto e'$
$e\ \mathsf{val}$

$$\textsc{Eval-Lam}\ \frac{}{\lambda x.e\ \mathsf{val}} \qquad \textsc{Eval-App}\ \frac{e_1 \mapsto \lambda x.e_1'}{e_1 e_2 \mapsto (\!|e_2/x|\!) e_1'} \qquad \textsc{Eval-Var}\ \frac{}{x[\overline{e}]\ \mathsf{val}}$$

$$\textsc{Eval-Zero}\ \frac{}{\mathsf{Z}\ \mathsf{val}} \qquad \textsc{Eval-Suc-1}\ \frac{e\ \mathsf{val}}{\mathsf{S}(e)\ \mathsf{val}} \qquad \textsc{Eval-Suc-2}\ \frac{e \mapsto e'}{\mathsf{S}(e) \mapsto \mathsf{S}(e')}$$

$$\textsc{Eval-Case-1}\ \frac{e_1 \mapsto \mathsf{Z}}{\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(x) \Rightarrow e_3 \mapsto e_2}$$

$$\textsc{Eval-Case-2}\ \frac{e_1 \mapsto^* \mathsf{S}(n) \quad \mathsf{S}(n)\ \mathsf{val}}{\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(x) \Rightarrow e_3 \mapsto (\!|n/x|\!) e_3}$$

$$\textsc{Eval-Let}\ \frac{e_1 \mapsto^* e_1' \quad e_1'\ \mathsf{val}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mapsto (\!|e_1'/x|\!) e_2}$$

Figure 2. RSLC small-step evaluation rules. They're exactly the same as for any other call-by-name λ-calculus, so don't bother reading them. Why am I even including them?

Ψ or however you happen to swing), because no substitution has happened yet, but after n "λx"s have been β-reduced, the $\overline{e}$ list will have n expressions in it. It might help to think of it like quantum superposition, I guess?

Anyway, Figure 2 shows the evaluation rules, which are basically what you'd expect, and Figure 3 shows the substitution rules, which are new. In the normal λ-calculus, substitution would stop when a new binding's name is the same as the one being substituted for.

However, in RSLC, as shown in Subst-Capture, we switch to a second substitution mode, Capture, in which substituted expressions are appended to a variable's $\overline{e}$ list (in the Capture-Var-X rule). Similarly, when substituting for x before switching to Capture mode (in the Subst-Var-X rule), we nondeterministically select an expression from x's $\overline{e}$ list to replace x with (with uniform randomness).

$(\!|e_0/x|\!) e$

$$\textsc{Subst-Capture}\ \frac{}{(\!|e_0/x|\!) \lambda x.e = \lambda x.\{\!|e_0/x|\!\} e} \qquad \textsc{Subst-Lam}\ \frac{y \neq x}{(\!|e_0/x|\!) \lambda y.e = \lambda y.(\!|e_0/x|\!) e'}$$

$$\textsc{Subst-Var-X}\ \frac{e_1 \in (e_0, (\!|e_0/x|\!)\overline{e})}{(\!|e_0/x|\!) x[\overline{e}] = e_1} \qquad \textsc{Subst-Var-Y}\ \frac{y \neq x}{(\!|e_0/x|\!) y[\overline{e}] = y[(\!|e_0/x|\!)\overline{e}]}$$

$\{\!|e_0/x|\!\} e$

$$\textsc{Capture-Var-X}\ \frac{}{\{\!|e_0/x|\!\} x[\overline{e}] = x[e_0, \{\!|e_0/x|\!\}\overline{e}]} \qquad \textsc{Capture-Var-Y}\ \frac{y \neq x}{\{\!|e_0/x|\!\} y[\overline{e}] = y[\{\!|e_0/x|\!\}\overline{e}]}$$

Figure 3. Selected RSLC substitution rules. The "$e_1 \in$ ..." premise in Subst-Var-X represents nondeterministic choice.

2.1 Evaluation Example

To demonstrate, here's the evaluation of $(\lambda x.(\lambda x.x))\ A\ B$ from the introduction:

$$
\begin{array}{rll}
 & (\lambda x.(\lambda x.x[]))\ A\ B & \\
\mapsto & ((\!|A/x|\!)(\lambda x.x[]))\ B & \textsc{Eval-App} \\
= & (\lambda x.\{\!|A/x|\!\} x[])\ B & \textsc{Subst-Capture} \\
= & (\lambda x.x[A])\ B & \textsc{Capture-Var-X} \\
\mapsto & (\!|B/x|\!) x[A] & \textsc{Eval-App} \\
= & \textit{(any element of the list } [B, A]) & \textsc{Subst-Var-X}
\end{array}
$$

Now the fairness to A we wanted in the intro is restored.

3. Random Programming with RSLC

Already we have a system with which we can implement simple random number generators, such as a simple 6-sided die:[1]

$$(\lambda x.(\lambda x.(\lambda x.(\lambda x.(\lambda x.x))))))\ \boxdot\ \boxdot\ \boxdot\ \boxdot\ \boxdot\ \boxdot$$

However, this isn't really "programming", and Sully thought for sure I wouldn't be able to do anything useful with this language, which of course I wasn't going to stand by. The first thing I wanted to do was to recursively generate a random natural number, something like this:

$$\mathsf{Y}\ (\lambda f.\lambda n.(\lambda x.\lambda x.x)\ n\ (f\ \mathsf{S}(n)))\ \mathsf{Z} \tag{1}$$

This term would randomly choose whether to output n, its argument, or to call itself with $\mathsf{S}(n)$. However, the conventional Y-combinator doesn't work as intended in RSLC, because once a function is substituted into itself for its first

[1] Really I just wanted an excuse to typeset LaTeX dice.

argument, subsequent arguments will be captured in the substituted version. Even ignoring the internal workings of the combinator, and supposing some Y that satisfies $\mathsf{Y}\ g \mapsto^* g\ (\mathsf{Y}\ g)$, here's what happens:

$$\begin{array}{ll} & \mathsf{Y}\ (\lambda f.\lambda n.(\lambda x.\lambda x.x)\ n\ (f\ \mathsf{S}(n)))\ \mathsf{Z} \\ \mapsto^* & (\lambda n.(\lambda x.\lambda x.x)\ n\ (\mathsf{Y}\ (\lambda f.\lambda n.\ldots n\ldots)\ \mathsf{S}(n)))\ \mathsf{Z} \\ \mapsto & (\!|\mathsf{Z}/n|\!)((\lambda x.\lambda x.x)\ n\ (\mathsf{Y}\ (\lambda f.\lambda n.\ldots n\ldots)\ \mathsf{S}(n))) \\ = & ((\lambda x.\lambda x.x)\ (\!|\mathsf{Z}/n|\!)n\ (\mathsf{Y}\ (\lambda f.\lambda n.\ldots.\{\mathsf{Z}/n\}n\ldots)\ \mathsf{S}((\!|\mathsf{Z}/n|\!)n))) \end{array}$$

But wait – I only wanted random-capture on x, not on n! As a result, though we intended 0 to be output with half probability, and 1 with quarter probability, and so on, actually 0 gets output substantially more often (5/8, I think).

3.1 The new Y combinator(s)

In order to properly recurse in RSLC, we need a new convention in which recursive functions get themselves substituted into them *last*, after all other arguments have been applied. The desired identity is $\mathsf{Y}_1\ g\ a \mapsto^* g\ a\ (\mathsf{Y}_1 g)$ for one-argument functions (and $\mathsf{Y}_2\ g\ a\ b \mapsto^* g\ a\ b\ (\mathsf{Y}_2\ g)$ for two-argument functions, and so on).

Unfortunately, I wasn't even able to find an RSLC term that satisfied that identity. I had to also alter the way recursive functions call themselves – by referring to the combinator itself, passing it the recursive arguments, and *then* the substituted version of themselves. Since that makes no sense in words, just have a look:

$$\begin{array}{rcl} \mathsf{Y}_1 & = & \lambda ag.\ gag \\ \mathsf{Y}_2 & = & \lambda abg.\ gabg \\ \mathsf{Y}_3 & = & \lambda abcg.\ gabcg \\ & \ldots & \end{array}$$

Here's a sort of handwavy syntactic transformation example for fix constructs that supports one, two, etc arguments:

$$\begin{array}{rcl} \mathsf{fix}_1(f,a.e) & = & \lambda a.\mathsf{Y}_1\ a\ (\lambda af.[^{(\mathsf{Y}_1\ x\ f)}/_{f(x)}]e) \\ \mathsf{fix}_2(f,a,b.e) & = & \lambda ab.\mathsf{Y}_1\ a\ b\ (\lambda abf.[^{(\mathsf{Y}_1\ x\ y\ f)}/_{f(x)(y)}]e) \\ & \ldots & \end{array}$$

Note that this transformation doesn't just substitute for the variable f in e but also reorders the arguments it's applied to. So now the term from Equation 1 becomes:

$$\mathsf{Y}_1\ \mathsf{Z}\ (\lambda n.\lambda f.(\lambda x.\lambda x.x)\ n\ (\mathsf{Y}_1\ \mathsf{S}(n)\ f)) \qquad (2)$$

And evaluates as intended:

$$\begin{array}{ll} \mapsto^* & (\lambda n.\lambda f.(\lambda x.\lambda x.x)\ n\ (\mathsf{Y}_1\ \mathsf{S}(n)\ f))\ \mathsf{Z}\ (\lambda n.\lambda f.\ldots.) \\ \mapsto & (\!|\mathsf{Z}/n|\!)\lambda f.(\lambda x.\lambda x.x)\ n\ (\mathsf{Y}_1\ \mathsf{S}(n)\ f))\ (\lambda n.\lambda f.\ldots.) \\ = & (\lambda f.(\lambda x.\lambda x.x)\ \mathsf{Z}\ (\mathsf{Y}_1\ \mathsf{S}(\mathsf{Z})\ f))\ (\lambda n.\lambda f.\ldots.) \\ \mapsto & (\lambda x.\lambda x.x)\ \mathsf{Z}\ (\mathsf{Y}_1\ \mathsf{S}(\mathsf{Z})\ (\lambda n.\lambda f.\ldots.)) \end{array}$$

Figure 4 shows the experimental results of evaluating this term 10,000 times.

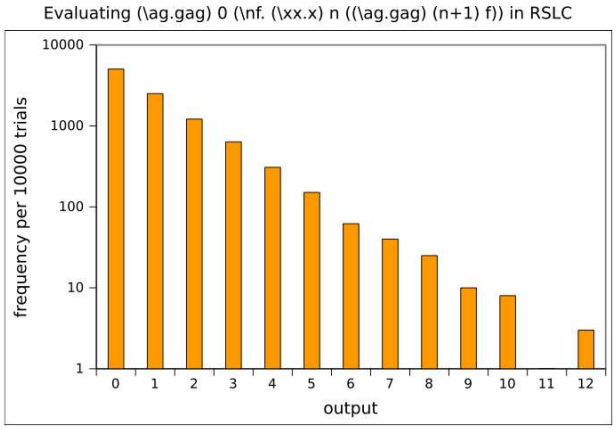

Figure 4. Computing random numbers with the term in Equation 2. Log scale, lower is better. (I couldn't show 13 because it showed up 0 times, and $\log 0 = -\infty$.)

3.2 Uniform random number generation

The next thing I wanted to do was compute random numbers with a uniform distribution. Previously we were just selecting between two possible bindings for x, statically defined in the program text. I wondered, "can I dynamically generate a term with n shadow-bindings[2] that causes a variable to have n expressions in its $\overline{e}$ list?" Something like:

$$(\lambda x.(\lambda x.\ldots.(\lambda x.x)))\ 0\ 1\ 2\ \ldots\ n$$

Of course this was where I had to iron out all the crazy parts in the previous sections, but since I already presented it in a manner consistent with it ain't being no thang, I'm going to make this part look easy:

$$\begin{array}{rcl} \rho & = & \lambda \mathbf{t}.\lambda n. \\ & & \quad \mathsf{case}\ n\ \mathsf{of} \\ & & \quad\quad \mathsf{Z} \Rightarrow \lambda f.\ \mathbf{t}\ \mathsf{Z} \\ & & \quad\quad \mathsf{S}(n_2) \Rightarrow \lambda f.\ \mathsf{Y}_2\ ((\lambda x.\mathbf{t})\ n)\ n_2\ f \end{array}$$

$$\mathsf{rand}[0,N] \quad = \quad \mathsf{Y}_2\ (\lambda x.x)\ N\ \rho$$

The variable $\mathbf{t}$ accumulates "λx"s as ρ recurses (in the $\mathsf{S}(n_2)$ case). You might ask: why does the λf need to be inside the case statement? Because otherwise, in Eval-Case-2, n_2 would get captured in the thing substituted for f.

A brief, very condensed, evaluation run-through:

$$\begin{array}{rcl} \mathsf{rand}[0,2] & = & \mathsf{Y}_2\ (\lambda x.x)\ 2\ \rho \\ & \mapsto^* & \mathsf{Y}_2\ ((\lambda x.(\lambda x.x))\ 2)\ 1\ \rho \\ & \mapsto^* & \mathsf{Y}_2\ ((\lambda x.(\lambda x.(\lambda x.x))\ 2)\ 1)\ 0\ \rho \\ & \mapsto^* & ((\lambda x.(\lambda x.(\lambda x.x))\ 2)\ 1)\ 0 \\ & \mapsto^* & (\!|0/x|\!)x[1,2] \end{array}$$

[2] Shadow binding: Sor/Wiz 3, Complete Arcane. Will save negates.

4. Deterministic Programming with RSLC

Got here. Up until now I showed how random scope adds inherent nondeterminism that can be harnessed to simulate dice and make silly graphs. The other thing Sully didn't believe would be possible was writing deterministic RSLC programs that *avoid* using "the feature".

In the last section I already introduced the capture-free Y combinator(s). Implementing addition using that is no trouble:

$$
\begin{array}{lcl}
\maltese & = & \lambda m.\lambda n. \\
 & & \quad \mathsf{case}\ m\ \mathsf{of} \\
 & & \quad\quad \mathsf{Z} \quad\Rightarrow\quad \lambda f.\, n \\
 & & \quad\quad \mathsf{S}(m') \quad\Rightarrow\quad \lambda f.\, \mathsf{S}(\mathsf{Y}_2\ m'\ n\ f)
\end{array}
$$

$$\mathsf{add}\ M\ N \quad = \quad \mathsf{Y}_2\ M\ N\ \maltese$$

But it turns out that multiplication, which uses addition, winds up needing the eager let construct. Among the several ways to implement it fully lazily, some introduce accidental randomness by letting some of times's variables get captured, and others simply cause my implementation to run out of stack space or start swapping to disk when evaluating. Uh, anyway, here it is.

$$
\begin{array}{lcl}
\bigstar & = & \lambda \hat{m}.\lambda \hat{n}. \\
 & & \quad \mathsf{case}\ \hat{m}\ \mathsf{of} \\
 & & \quad\quad \mathsf{Z} \quad\Rightarrow\quad \lambda \hat{f}.\, \mathsf{Z} \\
 & & \quad\quad \mathsf{S}(\hat{m}') \quad\Rightarrow\quad \lambda \hat{f}. \\
 & & \quad\quad\quad \mathsf{let} \quad \hat{x} = \mathsf{Y}_2\ \hat{m}'\ \hat{n}\ \hat{f} \\
 & & \quad\quad\quad \mathsf{in} \quad \mathsf{add}\ \hat{x}\ \hat{n}
\end{array}
$$

$$\mathsf{times}\ M\ N \quad = \quad \mathsf{Y}_2\ M\ N\ \bigstar$$

Note that since times refers to add, times's variable names must have different names. I put hats on them.

Here's the factorial function:

$$
\begin{array}{lcl}
! & = & \lambda \mathring{n}. \\
 & & \quad \mathsf{case}\ \mathring{n}\ \mathsf{of} \\
 & & \quad\quad \mathsf{Z} \quad\Rightarrow\quad \lambda \mathring{f}.\, \mathsf{S}(\mathsf{Z}) \\
 & & \quad\quad \mathsf{S}(\mathring{n}') \quad\Rightarrow\quad \lambda \mathring{f}. \\
 & & \quad\quad\quad \mathsf{let} \quad \mathring{x} = \mathsf{Y}_1\ \mathring{n}'\ \mathring{f} \\
 & & \quad\quad\quad \mathsf{in} \quad \mathsf{times}\ \mathring{n}\ \mathring{x}
\end{array}
$$

$$\mathsf{fact}\ N \quad = \quad \mathsf{Y}_1\ N\ !$$

To compute fibonacci numbers, I used the standard Church encoding for pairs:

$$
\begin{array}{rcl}
(e_1, e_2) & = & \lambda b.\, b\ e_1\ e_2 \\
\mathsf{fst}\ e & = & e(\lambda x.\lambda y.x) \\
\mathsf{snd}\ e & = & e(\lambda x.\lambda y.y)
\end{array}
$$

Hence:

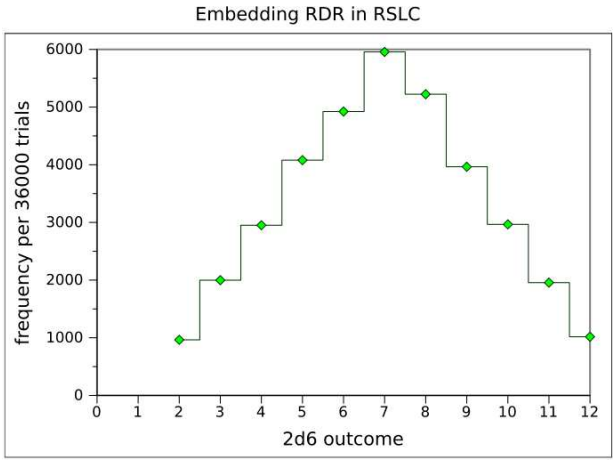

Figure 5. RSLC is a generalization of the Random Distance Run [RDR12].

$$
\begin{array}{lcl}
\Phi & = & \lambda \tilde{n}. \\
 & & \quad \mathsf{case}\ \tilde{n}\ \mathsf{of} \\
 & & \quad\quad \mathsf{Z} \quad\Rightarrow\quad \lambda \tilde{f}.\, (\mathsf{Z},\, \mathsf{S}(\mathsf{Z})) \\
 & & \quad\quad \mathsf{S}(\tilde{n}') \quad\Rightarrow\quad \lambda \tilde{f}. \\
 & & \quad\quad\quad \mathsf{let} \quad \tilde{p} = \mathsf{Y}_1\ \tilde{n}'\ \tilde{f} \\
 & & \quad\quad\quad\quad\quad\ \tilde{x}_1 = \mathsf{fst}\ \tilde{p} \\
 & & \quad\quad\quad\quad\quad\ \tilde{x}_2 = \mathsf{snd}\ \tilde{p} \\
 & & \quad\quad\quad \mathsf{in} \quad (\tilde{x}_2,\, \mathsf{add}\ \tilde{x}_1\ \tilde{x}_2)
\end{array}
$$

$$\mathsf{fib}\ N \quad = \quad \mathsf{Y}_1\ N\ \Phi$$

At this point I could include some addition and multiplication tables like the kind you saw in 3rd grade, but it would be no surprise. Suffice to say that it works.

5. Applications

Now that I showed in an abstract sense how RSLC can be used to express popular computations from conventional lambda calculus as well as some novel random programming techniques, let's do a real-world application. Using the uniform combinator from Section 3.2, we can make standard dice for any nonzero number of sides:

$$
\begin{array}{rcl}
\mathsf{d}(\mathsf{S}(N)) & = & \mathsf{add}\ 1\ \mathsf{rand}[0,N] \\
\mathsf{2d}(\mathsf{S}(N)) & = & \mathsf{let}\ x = \mathsf{add}\ \mathsf{rand}[0,N]\ \mathsf{rand}[0,N] \\
 & & \mathsf{in}\ \mathsf{add}\ 2\ x \\
\mathsf{3d}(\mathsf{S}(N)) & = & \mathsf{let}\ x = \mathsf{add}\ \mathsf{rand}[0,N]\ \mathsf{rand}[0,N] \\
 & & \mathsf{in}\ \mathsf{let}\ y = \mathsf{add}\ x\ \mathsf{rand}[0,N] \\
 & & \mathsf{in}\ \mathsf{add}\ 3\ y
\end{array}
$$

For $N = 5$, using the $\mathsf{2d}(\mathsf{S}(N))$ combinator (more commonly known as 2d6), we obtain a random distribution equivalent to the one employed in the Random Distance Run [RDR12], as shown in Figure 5.

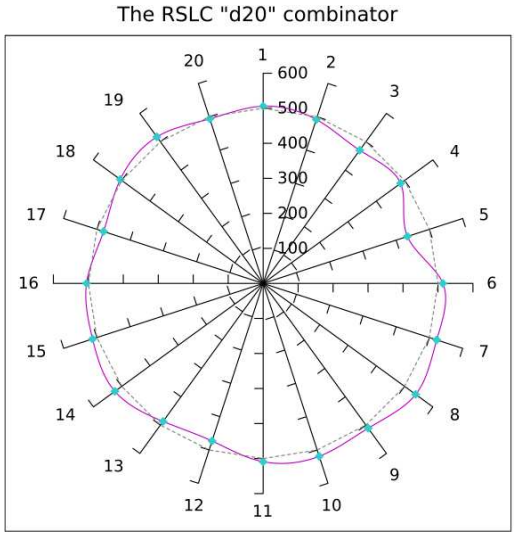

Figure 6. RSLC can also be used to implement popular role-playing games.

The value $N = 19$ yields the d20 combinator, which generates the random distribution shown in Figure 6 above. Prior work has shown this combinator to be quite useful in tabletop roleplaying games [Gyg77, Gyg78, Gyg79].

6. Theoretical Concerns

Some programming language theoreticians will speak of the so-called "Frame Rule", which expresses abstraction: if a property P holds for a certain function f, then in another function g which invokes x, if $P(x)$ implies $P(g)$, then $P([f/x]g)$.

In the context of RSLC, we scoff at this rule.

7. Concluding Remarks

RSLC is an extension to standard un(i)typed lambda calculus in which substitution is essentially completely broken. However, since substitution is broken in a particularly careful way, it is possible to write interesting programs that make use of the language's built-in nondeterminism. Furthermore, I showed that it is still possible to write a bunch of deterministic programs that are commonly used as examples/proofs-of-concept, as long as you put silly hats on some of your variables.

I believe you could extend RSLC's substitution schema to a typed lambda calculus, and furthermore if it only substitutes for variables of the same type, it would even be type-safe.

My implementation of RSLC in Haskell is available at `https://github.com/bblum/sigbovik/blob/master/RSLC/RSLC.hs`.

Acknowledgement

Michael Sullivan is, of course, the "Sully" that I mentioned (and will in the appendix continue to mention).

References

[707] Tom Murphy 7. Wikiplia: The free programming language that anyone can edit. In *Proceedings of the 1st Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q. Bovik's 2^6th birthday*, 2007.

[Blu12] Ben Blum. A randomized commit protocol for adversarial - nay, supervillain - workloads. In *Proceedings of the 6th Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q. Bovik's 2^6th birthday*, 2012.

[BR99] David Beazley and Guido Van Rossum. *Python; Essential Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 1999.

[Chu85] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton University Press, Princeton, NJ, USA, 1985.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 236–249, New York, NY, USA, 2003. ACM.

[Goo12] Google. The Go programming language specification. `http://golang.org/ref/spec`, 2012. With the Go Team.

[Gyg77] Gary Gygax. *Advanced Dungeons & Dragons: Monster Manual*. TSR, 1977.

[Gyg78] Gary Gygax. *Advanced Dungeons & Dragons: Player's Handbook*. TSR, 1978.

[Gyg79] Gary Gygax. *Advanced Dungeons & Dragons: Dungeon Master's Guide*. TSR, 1979.

[Ha11] Richard Ha. MLA-style programming. In *Proceedings of the 5th Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q. Bovik's 2^6th birthday*, 2011.

[Hoa12] Graydon Hoare. Rust reference manual. `http://dl.rust-lang.org/doc/rust.html`, 2012. With the Rust Team.

[Pey03] Simon Peyton Jones. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1), Jan 2003. `http://www.haskell.org/definition/`.

[RDR12] Wolfgang Richter, CMU Computer Science Department, and Random Structures and Algorithms. The random distance run. `http://www.cs.cmu.edu/~RDR/`, 2012.

[SV12] Robert J. Simmons and Tom Murphy VII. A modest proposal for the purity of programming. In *Proceedings of the 6th Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q. Bovik's 2^6th birthday*, 2012.

A. Discussion of Substitution Rules

Section 2 shows some "most important" example substitution rules, saving the complete set of rules for the appendix, which is where we are now, hence Figure 7 below.

Now, Sully thought my two-judgement scheme was gross, and proposed his own, which I think is operationally equivalent. Instead of switching to a different judgement the first time you traverse a matching binder (and annotating variables with expression-lists), in you always carry around a counter, which you increment when crossing matching binders and which represents a "substitution probability" (really the inverse thereof).

Figure 8 shows rules for this alternate substitution scheme. In these rules, the notation $[\![e_0/x]\!]^p\, e$ represents substituting with probability $1/p$, and the default substitution used by the evaluation rules, $(\!|e_0/x|\!)\, e$, is equivalent to substituting with probability 1, i.e., $[\![e_0/x]\!]^1\, e$. Also, of course, variables don't have lists attached; they are x rather than $x_{\overline{e}}$. While this alternate system eliminates the clunkiness of annotating variables, it has its own clunkiness because the rules Subst2-Var-X-Yes and Subst2-Var-X-No must have explicit probabilities for which of the two of them will apply.

$\boxed{(\!|e_0/x|\!)\, e}$

$$\textsc{Subst-Lam-Capture}\ \frac{}{(\!|e_0/x|\!)\lambda x.e = \lambda x.\{\!|e_0/x|\!\}e} \qquad \textsc{Subst-Lam}\ \frac{y \neq x}{(\!|e_0/x|\!)\lambda y.e = \lambda y.(\!|e_0/x|\!)e'} \qquad \textsc{Subst-App}\ \frac{}{(\!|e_0/x|\!)e_1 e_2 = (\!|e_0/x|\!)e_1 (\!|e_0/x|\!)e_2'}$$

$$\textsc{Subst-Var-X}\ \frac{e_1 \in (e_0, (\!|e_0/x|\!)\overline{e})}{(\!|e_0/x|\!)x[\overline{e}] = e_1} \qquad \textsc{Subst-Var-Y}\ \frac{y \neq x}{(\!|e_0/x|\!)y[\overline{e}] = y[(\!|e_0/x|\!)\overline{e}]}$$

$$\textsc{Subst-Case-Capture}\ \frac{}{(\!|e_0/x|\!)\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(x) \Rightarrow e_3 = \mathsf{case}\ (\!|e_0/x|\!)e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow (\!|e_0/x|\!)e_2;\ \mathsf{S}(x) \Rightarrow \{\!|e_0/x|\!\}e_3}$$

$$\textsc{Subst-Case}\ \frac{y \neq x}{(\!|e_0/x|\!)\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(y) \Rightarrow e_3 = \mathsf{case}\ (\!|e_0/x|\!)e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow (\!|e_0/x|\!)e_2;\ \mathsf{S}(y) \Rightarrow (\!|e_0/x|\!)e_3}$$

$$\textsc{Subst-Zero}\ \frac{}{(\!|e_0/x|\!)\mathsf{Z} = \mathsf{Z}} \qquad \textsc{Subst-Suc}\ \frac{}{(\!|e_0/x|\!)\mathsf{S}(e) = \mathsf{S}((\!|e_0/x|\!)e)}$$

$$\textsc{Subst-Let-Capture}\ \frac{}{(\!|e_0/x|\!)\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 = \mathsf{let}\ x = (\!|e_0/x|\!)e_1\ \mathsf{in}\ \{\!|e_0/x|\!\}e_2} \qquad \textsc{Subst-Let}\ \frac{y \neq x}{(\!|e_0/x|\!)\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2 = \mathsf{let}\ y = (\!|e_0/x|\!)e_1\ \mathsf{in}\ (\!|e_0/x|\!)e_2}$$

$\boxed{\{\!|e_0/x|\!\}e}$

$$\textsc{Capture-Lam}\ \frac{}{\{\!|e_0/x|\!\}\lambda y.e = \lambda y.\{\!|e_0/x|\!\}e'} \qquad \textsc{Capture-App}\ \frac{}{\{\!|e_0/x|\!\}e_1 e_2 = \{\!|e_0/x|\!\}e_1 \{\!|e_0/x|\!\}e_2'} \qquad \textsc{Capture-Var-X}\ \frac{}{\{\!|e_0/x|\!\}x[\overline{e}] = x[e_0, \{\!|e_0/x|\!\}\overline{e}]} \qquad \textsc{Capture-Var-Y}\ \frac{y \neq x}{\{\!|e_0/x|\!\}y[\overline{e}] = y[\{\!|e_0/x|\!\}\overline{e}]}$$

$$\textsc{Capture-Case}\ \frac{}{\{\!|e_0/x|\!\}\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow e_2;\ \mathsf{S}(y) \Rightarrow e_3 = \mathsf{case}\ \{\!|e_0/x|\!\}e_1\ \mathsf{of}\ \mathsf{Z} \Rightarrow \{\!|e_0/x|\!\}e_2;\ \mathsf{S}(y) \Rightarrow \{\!|e_0/x|\!\}e_3}$$

$$\textsc{Capture-Zero}\ \frac{}{\{\!|e_0/x|\!\}\mathsf{Z} = \mathsf{Z}} \qquad \textsc{Capture-Suc}\ \frac{}{\{\!|e_0/x|\!\}\mathsf{S}(e) = \mathsf{S}(\{\!|e_0/x|\!\}e)} \qquad \textsc{Capture-Let}\ \frac{}{\{\!|e_0/x|\!\}\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2 = \mathsf{let}\ y = \{\!|e_0/x|\!\}e_1\ \mathsf{in}\ \{\!|e_0/x|\!\}e_2}$$

Figure 7. Complete RSLC substitution and capture rules.

Lambda calculus (also written as Î»-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It is a universal model of computation that can be used to simulate any Turing machine. It was introduced by the mathematician Alonzo Church in the 1930s as part of his research into the foundations of mathematics.