Seminar Report

Unlambda

Franziska Rapp
franziska.rapp@student.uibk.ac.at

15 February 2013

Supervisor: Cynthia Kop

Abstract

This report gives an introduction to the esoteric programming language `Unlambda`. The focus is on outlining the similarities to more well-known languages. In order to understand the language, some example programs are analyzed. Furthermore, it is described how to transform programs from `lambda calculus` into `Unlambda`.

Contents

1 Introduction

`Unlambda` is a functional and esoteric programming language invented by David Madore in 1999. It is an implementation of the `lambda calculus` without any λ; therefore the name `Unlambda`. The following two quotations give a good insight into what you can expect with the programming language `Unlambda`.

> *"Unlambda is meant as a demonstration of very pure functional programming rather than for practical use."*
>
> Author unknown

> *"Debugging or reading Unlambda programs is just about impossible."*
>
> Madore

So `Unlambda` is not the language to develop new software with. However, it is still an interesting language in terms of functionality and expressive power. The language uses solely functions and function application; not even variables are allowed. As the inventor of the language states: even an expert has trouble understanding `Unlambda` code. Although it is possible to evaluate an `Unlambda` program manually this requires a lot of patience and time. There are many interpreters[1] for this purpose written in highly diverse languages like `C`, `Java`, `Haskell`, `OCaml`, `Scheme`, `INTERCAL` and even one in `Unlambda`.

Coming back to the expressive power `Unlambda`, is a "Turing tarpit". This term is used differently in literature but they all have in common that a Turing tarpit is a programming language or system that is Turing complete. Moreover some sources demand for a minimal set of operations (so no syntactic sugar is tolerated) whereas others state that such a language or system is extremely slow calculating common tasks. The latter can be caused by a very small set of operations as well as unusual behaviour of functions which makes it really difficult to program with such a language. *"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."* [5] I will use the latter definition below because in terms of the first one, `Unlambda` would not be a Turing tarpit as there are functions which can be represented by others as described in section 3. The term "esoteric" is often used equivalently even if the Turing completeness is not officially required.

2 Syntax and Properties

`Unlambda` is based on combinatory logic which was invented by Moses Schönfinkel 1924 and later revisited by Haskell Curry. The combinatory logic operates with so called combinators and does not need any variables just as `Unlambda`. For evaluation strategy the eager evaluation is applied meaning that the arguments of a function are evaluated before applying the function to them.

[1] This is probably the smartest one: `http://inazz.jp/unlambda/`
It accepts also `Unlambda 2` programs.

Example 2.1. Eager evaluation:

$$\begin{aligned}&(2+3)*(4+7)\\ \rightarrow\ &5*(4+7)\\ \rightarrow\ &5*11\\ \rightarrow\ &55\end{aligned}$$

There is one additional rule for evaluation strategy described in section 3.

Moreover `Unlambda` provides neither data structures nor code structures, so writing your own functions is the only possibility. This requires some creativity because naturally there is more than one way to express things like booleans or numbers and the operations based on them. But it is not that easy to reuse user-defined functions because they can be neither saved nor named, since there are no variables.

`Unlambda` uses prefix notation, which means an operator is written in front of its operands. Infix notation cannot work because all operators are unary. Applying an operator to its operand is done by the apply operator `` ` ``, which is used instead of parentheses and works as follows:

$$`fx$$

This has the effect that the function f is applied to its argument x.

Every `Unlambda` program is a function, where functions are defined as follows:

- Every combinator is a function.
  (combinators will be explained in the next section)
- If X and Y are functions, then $`XY$ is also a function.

3 Combinators

A combinator is a higher order function which takes exactly one function as argument and returns a potentially different function. The combinators s, k and i are inherited from the combinatory logic and most frequently used. Although the i combinator can be expressed by the s and k combinators like $``skk$, using i shortens programs and their evaluation.

3.1 Currying

As mentioned above, a combinator takes exactly one argument which is applied by the apply operator. Since the restriction on only one argument is too hard, Moses Schönfinkel invented a technique, later called currying, which makes it possible to define functions with more than one argument. Therefore the function is applied consecutively to each of its arguments, returning always another function waiting for the rest of the needed arguments until they are all applied.

3.2 Combinator Semantics

The easiest combinator is probably the i which is nothing more than the identity function. It takes one argument and returns it.

$$`ix \to x$$

The k combinator can ignore an argument, which could be very helpful. It takes two arguments and returns the first one.

$$`kxy \to x$$

The last combinator of the combinatory logic is the s combinator which takes three arguments, returning the first applied to the third and the result of that applied to the result of the second applied to the third. Whereas in combinatory logic this rule could be written as follows

$$(((Sx)y)z) \to ((xz)(yz))$$

Transforming it into `Unlambda`, the opening parentheses are replaced by the apply operator and the closing ones are dropped. This yields

$$```sxyz \to ``xz`yz$$

Making use of the left associativity in combinatory logic almost every pair of parentheses can be dropped. Equivalently almost every apply operator could be dropped in `Unlambda`, which yields

$$sxyz \to xz`yz$$

However this is not common and so the left associativity is not used below. A very useful combinator for implementing booleans, especially $false$, is the v combinator. It takes one argument and returns v, so this is also an ignoring combinator like k.

$$`vx \to v$$

So far, printing characters is not possible. For this purpose there is the $.x$ combinator which works just as the identity function printing the character x as side effect. Here x could be any character.

$$`.xy \to y$$

Very similar to this combinator works the r combinator which is in fact just an abbreviation for $.\langle newline \rangle$.

The d combinator does not seem to be very interesting. It takes two arguments and returns the first applied to the second.

$$``dfx \to `fx$$

Nevertheless there is a special rule for this combinator. The second argument will be evaluated before the first. In particular, if there is no second argument, the first one would not be evaluated at all.

The following two combinators are strongly related to each other. The c combinator takes one argument and returns the application of this argument to the programs current state represented by the $\langle cont \rangle$ combinator.

$$`cx \to `x\langle cont \rangle$$

The $\langle cont \rangle$ combinator also takes one argument and when applied to that argument it sets the program to the state where the specific $\langle cont \rangle$ was created and replaces the creating term with the current argument of $\langle cont \rangle$. This behaviour probably becomes clearer with the first Example 4.1 of the next section.

Although it seems just about impossible the combinators s and k suffice for the Turing completeness of `Unlambda`.

4 Examples

Example 4.1. This shows the c and the $\langle cont \rangle$ combinator working together:

$``cik$	(c creates a $\langle cont \rangle$ applying i to this $\langle cont \rangle$)
$\to ``i\langle cont \rangle k$	(i returns $\langle cont \rangle$ as normal)
$\to `\langle cont \rangle k$	(applying $\langle cont \rangle$ takes us "back in time" ...)
$\to `kk$	(... and changes the original `ci to k)

The term $`kk$ is already in normal form because k needs two arguments.

Example 4.2. The count2 program of the `Unlambda` distribution prints n stars in each line, starting with 0 and looping forever. The terms evaluated next are in red boxes.

0. $``r`cd `.*`cd$
   $\to$ Save $``\boldsymbol{r}[\,]\ \boldsymbol{`.*`cd}$ for this $\langle cont \rangle$, where [ ] stands for a hole. [1]

1. $``r\ `d\ \langle cont \rangle\ `.*`cd$
   $\to$ Nothing can be done here; d is waiting for the second argument

2. $`\ `r`d\ \langle cont \rangle\ `.*`cd$
   $\to$ Print newline and return the argument $`d\langle cont \rangle$

3. $``d\ \langle cont \rangle\ `.*\ `cd$
   $\to$ Save $``\boldsymbol{d\langle cont \rangle`.*}\,[\,]$ for this $\langle cont \rangle$[2]

4. $``d\ \langle cont \rangle\ `.*\ `d\ \langle cont \rangle$
   $\to$ Nothing can be done here

5. $``d\ \langle cont \rangle\ `.*`d\ \langle cont \rangle$
   $\to$ Print $*$

6. $\backprime d$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$
   → d applies $\langle cont \rangle$ to $\backprime d \langle cont \rangle$

7. $\backprime$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$
   → Let Y be the current $\backprime d \langle cont \rangle$. Now the yellow $\langle cont \rangle$ is applied to Y. So we go back to step 0, putting Y in the hole.

8. $\backprime$ $\backprime r \backprime d$ $\langle cont \rangle$ $\backprime . * \backprime$ cd
   → Print newline

9. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime .*$ $\backprime cd$
   → Save $\backprime\backprime \boldsymbol{d \langle cont \rangle} \backprime \boldsymbol{. *}$ [ ] for this $\langle cont \rangle$[3]

10. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime .*$ $\backprime d$ $\langle cont \rangle$

11. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime . * \backprime$ d $\langle cont \rangle$
    → Print $*$

12. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$

13. $\backprime$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$
    → Now we go back to step 3, putting the $\backprime d \langle cont \rangle$ (which was created in step 9) in the hole.

14. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime . * d$ $\langle cont \rangle$
    → Print $*$

15. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$

16. $\backprime$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$
    → Now we go back to step 0, putting the $\backprime d \langle cont \rangle$ (which was created in step 9) in the hole.

17. $\backprime$ $\backprime r \backprime d$ $\langle cont \rangle$ $\backprime . * \backprime$ cd
    → Print newline

18. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime .*$ $\backprime cd$
    → Save $\backprime\backprime \boldsymbol{d \langle cont \rangle} \backprime \boldsymbol{. *}$ [ ] for this $\langle cont \rangle$[4]

19. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime .*$ $\backprime d$ $\langle cont \rangle$

20. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime . * \backprime$ d $\langle cont \rangle$
    → Print $*$

21. $\backprime\backprime d$ $\langle cont \rangle$ $\backprime d$ $\langle cont \rangle$

22. $\texttt{`}\ \langle cont\rangle\ \texttt{`}d\ \langle cont\rangle$
→ Now we go back to step 9, putting the $\texttt{`}d\langle cont\rangle$ (which was created in step 18) in the hole.

23. $\texttt{``}d\ \langle cont\rangle\ \texttt{`}.*\texttt{`}\ d\ \langle cont\rangle$
→ Print $*$

24. $\texttt{``}d\ \langle cont\rangle\ \texttt{`}d\ \langle cont\rangle$

25. $\texttt{`}\ \langle cont\rangle\ \texttt{`}d\ \langle cont\rangle$
→ Now we go back to step 3, putting the $\texttt{`}d\langle cont\rangle$ (which was created in step 18) in the hole.

26. $\texttt{``}d\ \langle cont\rangle\ \texttt{`}.*\texttt{`}\ d\ \langle cont\rangle$
→ Print $*$

27. $\texttt{``}d\ \langle cont\rangle\ \texttt{`}d\ \langle cont\rangle$

28. $\texttt{`}\ \langle cont\rangle\ \texttt{`}d\ \langle cont\rangle$
→ Now we go back to step 0, putting the $\texttt{`}d\langle cont\rangle$ (which was created in step 18) in the hole.

29. $\texttt{`}\ \texttt{`}r\texttt{`}d\ \langle cont\rangle\ \texttt{`}.*\texttt{`}\ cd$
→ Print newline

At this point we have printed the first four lines. So the output is:

```
*
**
***
```

Example 4.3. There are some really good possibilities to represent booleans. The choice given by:

- i for $true$
- v for $false$

is probably the smartest way because i and v are built-in functions in `Unlambda`. The internal implementation of the input functions of `Unlambda 2` (see section 7) uses also this approach. As a consequence, the function `AND` can be written as i !

Proof. There are only 4 cases:

1. $\texttt{``}iii \to \texttt{`}ii \to i$ $\qquad (true\ \texttt{AND}\ true \approx true)$
2. $\texttt{``}iiv \to \texttt{`}iv \to v$ $\qquad (true\ \texttt{AND}\ false \approx false)$

3. $\text{``}ivi \rightarrow \text{`}vi \rightarrow v \qquad (false \texttt{ AND } true \approx false)$

4. $\text{``}ivv \rightarrow \text{`}vv \rightarrow v \qquad (false \texttt{ AND } false \approx false)$

□

Another advantage is, that a boolean can be simply applied to an argument which should only be evaluated if the boolean is true. This is correct because v returns v (so the argument will be rejected), whereas i returns the argument which will be evaluated later on.

Apart from these advantages this choice for the booleans carries only disadvantages. The problem is that testing if a function is equal to v is almost impossible because of returning always itself. The only way to get along is to use continuations. For the functions `NOT`, `OR` and `IF-THEN-ELSE` is always a continuation needed. To show one example, the `IF-THEN-ELSE` function can be written as follows:

$$\text{``}s\text{`}kc\text{``}s\text{`}k\text{`}s\text{`}k\text{`}k\text{`}ki\,\text{``}ss\text{`}k\text{`}kk$$

With this function the code snippet "**if** x **then** y **else** z" can be written as follows:

$$\text{````}s\text{`}kc\text{``}s\text{`}k\text{`}s\text{`}k\text{`}k\text{`}ki\,\text{``}ss\text{`}k\text{`}kkxyz$$

where x, y and z are arbitrary terms.

In order to get an easier representation of the above mentioned functions the choice of k (as true) and $\text{`}ki$ (as false) would be the best one. With this approach the `IF-THEN-ELSE` function can simply be written as $i$, because $k$ is a function returning the first argument whereas $\text{`}ki$ is a function returning the second argument. So if the boolean of `IF-THEN-ELSE` is true, the first argument is returned, otherwise the second argument is returned as it should be.

5 Transforming Lambda Calculus into Unlambda

There is one relatively simple way to write `Unlambda` programs. Even though it is not innovative, transforming code similar to `lambda calculus` into `Unlambda` can be done systematically. This method is called *"abstraction elimination"* [4] since the lambda abstractions (not supported in `Unlambda`) get eliminated. This approach enables the use of variables, which makes programming much easier. Nevertheless, first of all such a code has to be written to make this transformation possible. Therefore a form of the `lambda calculus` without free variables is used which can also contain combinators. But normally they are not used because it is easier to write programs in pure `lambda calculus`. The special lambda terms used below are defined as follows:

- Every combinator is a lambda term.
- If x is a variable and Y is a lambda term, then λxY is a lambda term. The abstraction operator, λ, binds the variable x in the body of the abstraction.

- If x is a bound variable, then $\$x$ is a lambda term.
- If X and Y are lambda terms, then $`XY$ is also a lambda term.

Where $`$ is equivalent to an opening parenthesis in `lambda calculus` and the closing ones are dropped as explained in section 2. Because of the risk of confusion with the $.x$ combinator the point after $\lambda x$ is dropped. The \$ symbol indicates that the following symbol is a variable. Since this is just slightly modified `lambda calculus` code it is also possible to use existing programs as long as the functionality suffices.

Let F be an expression. There is always a lambda term λxF from which the λ should be removed. The process of "abstraction elimination" can be defined by induction on the structure of F. The following three cases have to be considered:

1. F is $\$x$. Then λxx has to be expressed which is exactly the identity function i.
2. F is a combinator or a variable other than $\$x$. Then F is a constant function which can be expressed as $`kF$ because applying an argument to $`kF$ the argument will be ignored as it is in `lambda calculus` if an argument is applied to a free variable.
3. F is an application of the form $`GH$ where the abstraction eliminations of $\lambda xG$ and $\lambda xH$ are already defined by induction. If an argument has to be applied to such an application in `lambda calculus`, the argument has to be applied to $G$ and to $H$ getting $H'$ and $G'$, respectively. The application of $H'$ to $G'$ would be the final result. In `Unlambda` this can be done via the $s$ combinator. So $`GH$ can be rewritten as $``s\,\lambda xG\,\lambda xH$ where λxG and λxH have to be simplified next. Applying an argument Z to this construct (leaving the lambda terms λxG and λxH as they are) yields
$$```s\,\lambda xG\,\lambda xH Z \rightarrow\ ``\lambda xGZ`\lambda xHZ$$
which is equivalent to what happens in `lambda calculus`.

Scanning a lambda term λxF from left to right, it suffice to consider the following three rules for "abstraction elimination" which can be derived from the the above explained cases (for the last one it suffice to write $``s$ because the arguments will be considered later on):

1. $` \rightarrow\ ``s$
2. $\$x \rightarrow\ i$
3. otherwise there is a combinator or a variable other than $\$x$
   $\rightarrow$ write $`k$ in front of the specific combinator/variable

Example 5.1. Consider the function $\lambda x`\$xk$, taking a function x and applying that function to k.

$$\lambda x`\$xk \quad \xrightarrow{\lambda-elimination} \quad ``si`kk$$

Here the first case applies for `, the second one for $\$x$ and the third one for k. The resulting function can be used applying it to an argument (in the following called z):

$$\begin{aligned} ```si`kkz &\rightarrow ``iz``kkz \\ &\rightarrow `z``kkz \\ &\rightarrow `zk \end{aligned}$$

As intended the argument z is now in the place of $\$x$ so the use of variables was successful.

If there are several lambdas in front of one expression, they could be eliminated incrementally, starting with the innermost and going outwards.

Example 5.2. Consider the function $\lambda x\lambda y\lambda z``\$z\$y\$x$ eliminating the lambda abstractions incrementally like that

$$\begin{aligned} \lambda x\lambda y\lambda z``\$z\$y\$x \quad &\xrightarrow{\lambda-elim.} \quad \lambda x\lambda y``s``si`k\$y`k\$x \\ &\xrightarrow{\lambda-elim.} \quad \lambda x``s``s`ks``s``s`ks`ki``s`kki``s`kk`k\$x \\ &\xrightarrow{\lambda-elim.} \quad ``s``s`ks``s``s`ks``s`kk`ks``s``s`ks``s``s`ks``s`kk`ks \\ &\qquad\qquad ``s`kk`ki``s``s`ks``s`kk`kk`ki``s``s`ks``s`kk`kk``s`kki \end{aligned}$$

Splitting the first step into smaller steps yields (terms that still have to be simplified are underlined):

$$\begin{aligned} \lambda x\lambda y\underline{\lambda z``\$z\$y\$x} &\rightarrow \lambda x\lambda y``s\underline{\lambda z`\$z\$y\$x} \\ &\rightarrow \lambda x\lambda y``s``s\underline{s\lambda z\$z\$y\$x} \\ &\rightarrow \lambda x\lambda y``s``si\underline{\lambda z\$y\$x} \\ &\rightarrow \lambda x\lambda y``s``si`k\$y\underline{\lambda z\$x} \\ &\rightarrow \lambda x\lambda y``s``si`k\$y`k\$x \end{aligned}$$

Obviously the length of the `Unlambda` term grows exponentially with factor 3 for each λ because of the ` operator which becomes $``s$, and also because each combinator and variable (except the variable captured by the current λ) is transformed into three symbols.

The method described above suffices for abstraction elimination. However, there are some tricks (also described in [4]) which make this technique much more efficient and counteracts the above described growth of the length. It can easily be seen that instead of rewriting v to $`kv$ the v can be left as it is. No

matter which argument $`kv$ or v, respectively, is applied to, the argument is evaluated and v is returned either way. It gets already more complex with the modification of rule 3 which can be applied more often. In fact it can be applied to any sub-expression F' of F in the term λxF (so not only for single symbols) if F' satisfies the following conditions:

1. F' does not involve x
2. The evaluation of F' terminates
3. No side effects during the evaluation of F'

The first condition is obvious. For the second and the third one it is necessary to know that if replacing F' by $`kF'$ the expression F' would be evaluated as soon as it is encountered, but this does not take into account that F' is eventually not applied to any argument. As a lambda term should only be evaluated if applied to an argument, this has to be considered also for the transformation. So this rule cannot be applied if F' does not terminate or has side effects like printing characters. Fortunately there is a workaround for the case of non-termination or side effects. Instead of going into F', it can be replaced by $`d`kF'$ delaying the evaluation of F' until the expression is applied to an argument.

Another shortcut can only be applied if the above mentioned conditions are satisfied. The lambda term $\lambda x`F'\$x$ can be rewritten as $F'$ considering the $\eta$-reduction of the `lambda calculus` [1]. If $F'$ does not terminate or produces side effects during evaluation it has to be replaced with $`dF'$ instead, with the same reasons as above.

6 Similarities to other Languages

As argued in the last section, `Unlambda` and `lambda calculus` are closely related. However the `lambda calculus` is not a real programming language but rather used in proof theory and theory of computation. It is equivalent to Turing machines, probably the most important computational model.

`Unlambda` combines the functionality of functional programming languages with the complications of esoteric programming languages. Representatives of this category are for example `INTERCAL`, `Befunge` and `brainfuck`. Whereas `INTERCAL` was the first and still canonical example of esoteric programming languages, `brainfuck` is currently one of the most famous representatives of this class.

The class of functional programming languages contains for example `Scheme` (a `Lisp` dialect), `OCaml`, `Haskell` and `Clojure`. `Scheme` has a function called "call-to-current-continuation" (abbreviated as call/cc). *"When the continuation procedure is invoked, it returns its argument to the continuation of the call to call/cc that created it."* [3] So the "continuation" is similar to $\langle cont \rangle$ in `Unlambda` and the call/cc function is similar to the c combinator. Therefore `Scheme` is in the most interesting part similar to `Unlambda`. There is also a `Scheme` to `Unlambda` compiler[2].

[2] E.g. available at `http://www.complang.tuwien.ac.at/schani/oldstuff/index.html`

There exists also a programming style used by some functional programming languages called "Continuation Passing Style" (CPS) which allows users to implement the call/cc function. Every function has an additional argument called continuation to which the result of the function is passed. Therefore a function will never return but invoke a continuation which will possibly invoke another function with another continuation later on terminating by a final continuation which can be the identity function. CPS enables also the implementation of exceptions, co-routines, threads and some other constructs. `Haskell` is one of the functional programming languages supporting directly CPS with Control.Monad.Cont [2].

The technique of currying is used in `lambda calculus` as well as in several functional programming languages like `Scheme` or `Haskell`.

Very similar to `Unlambda` are the languages `Iota` (its successor `Jot`) and `Lazy K` as they are both Turing tarpits and based on combinatory logic, like `Unlambda`. Moreover they are also esoteric programming languages as actually every language based on combinatory logic is. `Lazy K` was designed in order to get a programming language that does not touch the purity of the SKI combinators. Ben Rudiak-Gould, the inventor of `Lazy K`, states *"Unlambda isn't even close to being purely functional, because its I/O system depends crucially on side effects"* [6]. `Lazy K` uses input and output streams represented as lists of natural numbers as I/O system. `Iota` and `Jot` allow only two symbols, respectively. This is probably also the minimum number of symbols needed to get an unambiguous Turing complete programming language. Because if there was only one symbol, the "prefix property"[3] would not be satisfied and therefore the language is ambiguous.

7 Unlambda 2

The probably largest disadvantage of `Unlambda` is the lack of user interaction. This gap has been closed with `Unlambda 2` which introduces four new combinators. Three of them for user interaction and the last one for exiting a program.

- @ takes one argument (let it be x). A character is read from `STDIN`, stored as the "current character" and x is applied to i. If there are no characters available on `STDIN`, the current character remains undefined and $`xv$ is returned.

- $?u$ takes one argument (let it be x). The function returns $`xi$ if the current character is $u$, otherwise $`xv$ is returned.

- | takes one argument (let it be x). It returns $`x.u$ where $u$ is the current character. If there is no current character because the last application of @ was not successful, $`xv$ is returned.

- e takes one argument and exits the program with this argument as code.

[3] No code word is prefix of another code word.

8 Conclusion

As `Unlambda` is an esoteric programming language it is not sensible for present-day software development. Even though the continuation feature is very interesting, I do not think it will become standard, because it is very hard to program with more than one continuation and so the continuation is not sensible for "real languages". Also in `Scheme` it is very rarely used. The goal of unreadable code has been achieved. However, in minimalism or functional purity other languages like `Iota` or `Lazy K` are preferable. Even so, creative minds will get their money's worth because of the freedom of implementing data structures and code structures.

References

[1] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972. ISSN 1385-7258. doi: 10.1016/1385-7258(72)90034-0. URL `http://www.sciencedirect.com/science/article/pii/1385725872900340`.

[2] Andy Gill. Control.Monad.Cont. Technical Report 2.0.1.0, The University of Glasgow, nov 2010. URL `http://hackage.haskell.org/packages/archive/mtl/2.0.1.0/doc/html/Control-Monad-Cont.html`.

[3] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. *SIGPLAN Not.*, 25(6):66–77, jun 1990. ISSN 0362-1340. doi: 10.1145/93548.93554. URL `http://doi.acm.org/10.1145/93548.93554`.

[4] David Madore. The Unlambda Programming Language, aug 2001. URL `http://www.madore.org/~david/programs/unlambda/`.

[5] Alan J. Perlis. Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13, 1982.

[6] Ben Rudiak-Gould. Lazy K, 2002. URL `http://homepages.cwi.nl/~tromp/cl/lazy-k.html`.