ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp. 214-218
doi: 10.18178/wcse.2017.06.037

An analysis of computer programs using λ-calculus

Kittiphon Phalakarn+ and Athasit Surarerks

Department of Computer Engineering, Faculty of Engineering,
Chulalongkorn University, Thailand

Abstract. We propose that λ-calculus can be a mathematical model for analysing programs. The λ-calculus representation in this work includes numbers, Booleans, arithmetic operations, lists, functions, recursions, and loops. Moreover, we discuss whether a program can have a representation using simply typed λ-calculus. Since simply typed λ-calculus is not Turing complete, only some portions of a program can be represented. The automated λ-term evaluation process using its expression tree is also studied. For λ-term analysis, we propose an algorithm to maximize the number of parallel β-reductions done in each step, and review some literatures on methods to understand the semantics and flows of programs using their λ-terms. We believe that our results can be useful for analysing how a program can perform in parallel manner.

Keywords: analysis of computer programs, lambda calculus, simply typed lambda calculus, parallel

1. Introduction

Program analysis is an important activity in the field related to the theory of computation. One way to do program analysis is to consider its source code directly. However, this method may not be as simple as it sounds, because of the variations of syntax of programming languages.

Many techniques have been proposed in order to analyse and describe programs. One interesting technique is to characterize programs using some mathematical models. In this work, we focus on transformation technique between source code programs and (untyped) λ-calculus. It is known that the λ-calculus is a Turing complete mathematical model as shown in [1]. We are also interested in λ-calculus with type, called typed λ-calculus. These are the ways that a program must be expressed.

This paper aims to analyse computer programs using untyped and simply typed λ-calculus. We transform a program, which includes numbers, Booleans, arithmetic operations, lists, functions, recursions, and loops, into a λ-term. Then, we evaluate the term using an expression tree together with an automated program. We propose an algorithm to maximize the efficiency of the evaluation steps using parallelism on the expression tree. In addition, some literatures on λ-term analysis, such as semantics and flows analysis, are reviewed.

2. Preliminaries

2.1. Untyped λ-calculus

Untyped λ-calculus is a formal system invented by Alonzo Church in 1930s [2]. It describes how one-variable function works by substitution. The following definitions are adapted from definitions in [2].

Definition 1 (λ-terms). Given an infinite number of variables. The set of *λ-terms* is defined as follows.

1. All variables are λ-terms.
2. If M and N are any λ-terms, then (MN) is a λ-term (called an *application*).
3. If M is any λ-term and x is any variable, then (λx.M) is a λ-term (called an *abstraction*).

+ Corresponding author. Tel.: +66-87-594-3030
E-mail address: kittiphon.p@student.chula.ac.th

When we write λ-terms, we omit parentheses using association to the left convention. Abbreviations are MNPQ for (((MN)P)Q), λx.PQ for (λx.(PQ)), and $\lambda x_1 x_2 \ldots x_n.M$ for $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$. We use ≡ to show the syntactic identity of λ-terms, i.e., M ≡ N means that M is exactly the same λ-term as N. It is assumed that if MN ≡ PQ, then M ≡ P and N ≡ Q, and if λx.M ≡ λy.P, then x ≡ y and M ≡ P.

Definition 2 (Free and bound variables). For any subterm λx.M in a term P, M is called a *scope* of λx. An occurrence of a variable x in a term P is called

- *bound*, if it is in the scope of any λx in P,
- *bound and binding*, if it is the x in λx,
- *free*, otherwise. The set of all free variables of P is called FV(P).

Definition 3 (α-conversions). Let a term P contains a subterm λx.M, and let $y \notin FV(M)$. The act of replacing all x which is bound in the scope of the λx with y is called an *α-conversion* in P. If P can be changed to Q by finite steps of α-conversions, we say P *α-converts* to Q, or $P \equiv_\alpha Q$.

Definition 4 (β-redexes, β-reductions). Any term of a form (λx.M)N is called a *β-redex*. If a term P contains a subterm (λx.M)N, the act of replacing all free x within M with N (with some α-conversions to avoid clashes) to have P’ is called a *β-reduction* (notation $P \rhd_{1\beta} P'$). If P can be changed to Q by finite steps of β-reductions and α-conversions, we say P *β-reduces* to Q, or $P \rhd_\beta Q$.

Definition 5 (β-normal forms). A term Q which contains no β-redex is called a *β-normal form*. The set of all β-normal forms is called *β-nf*. If P β-reduces to Q in β-nf, then Q is called a β-normal form of P.

Definition 6 (β-equalities). We said P is *β-equal* to Q (notation $P =_\beta Q$) if Q can be obtained from P by finite steps of β-reductions, reversed β-reductions, and α-conversions. That is, $P =_\beta Q$ if there exist $P_0, \ldots, P_n$ $(n \geq 0)$ such that $P_0 \equiv P$, (for all $0 \leq i \leq n-1$)($P_i \rhd_{1\beta} P_{i+1}$ or $P_{i+1} \rhd_{1\beta} P_i$ or $P_i \equiv_\alpha P_{i+1}$), and $P_n \equiv Q$.

2.2. Simply typed λ-calculus

In typed λ-calculus, types are assigned to λ-terms to restrict which terms can be applied together. The type system we consider is simple type, which is defined in [2] as follows.

Definition 7 (Types). Given a finite or infinite number of symbols called *atomic types*, we define *types* as follows.

1. Every atomic type is a type.
2. If σ and τ are types, then (σ → τ) is a type (called a *function type*).

Remark that a type α → β represents an abstraction with an input of type α, and an output of type β. In type expression, we usually omit parentheses using association to the right convention. Abbreviations are σ → τ for (σ → τ), ρ → σ → τ for (ρ → (σ → τ)), and $\sigma_1 \to \ldots \to \sigma_n \to \tau$ for $(\sigma_1 \to (\ldots \to (\sigma_n \to \tau)\ldots))$.

Definition 8 (Typed variables). Given an infinite number of untyped variables. We make *typed variables* x : τ by attaching type to untyped variables in such a way that each untyped variable receives only one type, and every type τ is attached to an infinite number of variables.

Definition 9 (Simply typed λ-terms). The set of *simply typed λ-terms* is defined as follows.

1. All typed variables x : τ are simply typed λ-terms of type τ.
2. If M : σ → τ and N : σ, then (MN) : τ is a simply typed λ-term of type τ.
3. If x : σ and M : τ, then (λx.M) : σ → τ is a simply typed λ-term of type σ → τ.

3. Representing programs using λ-calculus

3.1. Numbers

One way of encoding numbers in λ-term is Church numerals, which is defined as follows.

Definition 10 (Church numerals). For every natural number n ≥ 0, the *Church numeral* for n is n’ defined by $n' \equiv \lambda xy.x^n y$ where $x^n y \equiv (x(x\,(\ldots\,(xy))))$ with n x’s.

Church numerals can also represent other types of data such as negative integers, floating-point numbers, and characters. Our main interest in this paper is only natural numbers.

3.2. Booleans

Church Boolean terms encode Boolean values **true** or **false** in λ-calculus. Consider the if-statement in Church Boolean term, **if** ≡ λctf.ctf where c corresponds to a Church Boolean, t corresponds to a statement when c is **true**, and f corresponds to a statement when c is **false**. The Boolean values can be implemented as **true** ≡ λtf.t and **false** ≡ λtf.f. Basic Boolean operations are

- **and** p q ≡ λpq.pqp
- **or** p q ≡ λpq.ppq
- **not** p ≡ λp.p(λtf.f)(λtf.t)
- **isZero** n ≡ λn.n(λz.(λtf.f))(λtf.t)

3.3. Arithmetic operations

All basic arithmetic operations can be expressed in λ-terms based on Church encodings. We use a recursion combinator **R** which has a property that, for all X, Y, k, **R**XY0' $=_\beta$ X and **R**XY(k+1)' $=_\beta$ Yk'(**R**XYk') (see [2] for more details). Some examples of basic arithmetic operations are

- **succ** n ≡ λnxy.nx(xy) or λnxy.x(nxy)
- **add** m n ≡ λmnxy.mx(nxy) or λmn.m **succ** n
- **mul** m n ≡ λmnxy.m(nx)y or λmn.m (**add** n) 0'
- **pred** n ≡ λn.**R**0'(λpq.p)n
- **sub** m n ≡ λmn.**R**m(λpq.**pred** q)n
- **eq** m n ≡ λmn.**isZero** (**add** (**sub** m n) (**sub** n m))

3.4. Lists

To represent lists, we need to encode **nil**, **cons**, **head**, and **tail** as shown below.

- **nil** ≡ λz.**true**
- **cons** x y ≡ λxyz.zxy
- **isNil** L ≡ λL.L(λxy.**false**)
- **head** L ≡ λL.L(λxy.x)
- **tail** L ≡ λL.L(λxy.y)

3.5. Functions, recursions, and loops

To encode functions, we use the following concept. The inputs of a function are indicated with λx in an abstraction (λx.M) and the outputs are expressed by its normal form after doing β-reductions. Recursive functions can be represented in λ-calculus using **Y** combinator with a property that, for all X, **Y**X $\rhd_\beta$ X(**Y**X) where **Y** combinator invented by Alan Turing is (λux.x(uux))(λux.x(uux)). Finally, to represent loops, we first transform the loop into a recursive function. We then use the same method to transform recursive functions into λ-terms.

3.6. Representing Programs

A program can also be considered as a sequence of functions. There are two approaches of structuring a program. First, we assume that the function is applied from outside to inside. The structure of the λ-term constructed in this way is $\lambda li_1.((\lambda li_2.(\lambda li_3\ldots)lo_2)lo_1)$ where li_i and lo_i are lists of inputs and outputs of the i-th statement. Another method of structuring a program is to apply variables from inside to outside. The idea of this method comes from 'list iteration' technique in [3]. The structure of the λ-term constructed in this way is $\lambda li.(f_k(\ldots(f_2(f_1\ li))))$ where li is a list of inputs, and f_j is a function of the j-th statement.

It is remarked that the first method does not strict its data structure to be a list. This technique is suitable for the second method only if the input and the output contain only one variable. In the other hand, a λ-term generated by the second method is shorter and more comprehensive than a λ-term from the first method.

A compiler, which compiles source code into a λ-term, could be constructed. An example of such compiler can be found in [4]. More literatures on representing programs using λ-calculus can be found in [5-7].

4. Representing programs using typed λ-calculus

In a simply typed λ-calculus, 'types' are added to prevent users from applying incorrect data to a function, e.g., **add true false** is an invalid typed λ-term. This makes the system close to a real programming language. For example, type of Church numerals $\lambda xy.x^n y$ is (α → α) → (α → α) ≡ **int**.

However, the simply typed system limits the system from being Turing complete as studied in [8-13]. Functions that can be expressed using simply typed λ-calculus are called 'extended polynomial'. To have Turing complete type system, we must consider a higher-order type, recursive type, type polymorphism, and typed atomic constant, which are out of the scope of this paper.

5. Evaluating programs in λ-calculus

After having a program in λ-term, we apply inputs to the term and do β-reductions until it gets into β-normal form. We use an automatic evaluator written in Python to parses the λ-term into an expression tree which includes variable nodes (x), application nodes (@), and abstraction nodes (λx) as shown in Fig. 1.

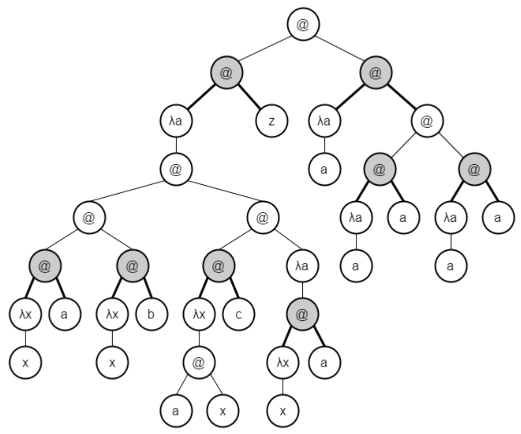

Fig. 1: An example of an expression tree. Redexes are shown in gray nodes.

```
Algorithm 1: MaxP(node $p$, set $V$, leftmost redex flag $f$)
if $p$ is variable node then
    return $(0,\ f)$
if $p$ is abstract node then
    return MaxP($p.child$, $V - \{p.var\}$, $f$)
if $p.left$ is abstract node and $V \cap FV(p.right) = \varnothing$ then
    if $f$ is true then
        $r$ = MaxP($p.left.child$, $V \cup \{p.left.var\}$, $false$)
        return $(r.first + 1,\ false)$
    else
        $r$ = MaxP($p.left.child$, $V \cup \{p.left.var\}$, $false$)
        $r1$ = MaxP($p.left$, $V$, $false$)
        $r2$ = MaxP($p.right$, $V$, $false$)
        $m = \max(r.first + 1, r1.first + r2.first)$
        return $(m,\ false)$
else
    $r1$ = MaxP($p.left$, $V$, $f$)
    $r2$ = MaxP($p.right$, $V$, $r1.second$)
    return $(r1.first + r2.first,\ r2.second)$
```

Fig. 2: Algorithm for computing the maximum number of reductions which can be done in parallel

The expression tree is then evaluated by doing β-reductions repeatedly. β-reduction is performed by searching the tree for a redex, duplicating the right subtree of the redex node, and replacing the appropriate nodes with this subtree. If there are more than one redexes, the leftmost outermost redex is needed to be reduced in order to ensure termination as proven in [14-16]. To further improve the performance of the β-reduction process, the idea of 'optimal reduction' is presented in many papers. Some papers use shared nodes to reduce the work of replacing subtree. Details can be found in [17-19].

6. λ-term analysis

6.1. Parallel reduction using expression tree

In Fig. 1, it might be more efficient if we can do all β-reductions simultaneously using parallelism. As discussed in [15], doing multiple β-reductions gives the same result as doing parallel β-reduction. However, some redexes are overlapped and cannot be done in parallel. Hence, the rules for our parallel β-reduction are

1. The leftmost redex needs to be reduced in order to ensure termination.
2. If we β-reduce an application node, we cannot β-reduce all redexes in its right subtree.
3. If we β-reduce an application node with 'λx' left child, we cannot do β-reduction at an application node in its left subtree where there is a bound x (in the scope of the 'λx' node) in the right subtree.

We propose Algorithm 1 (shown in Fig. 2) which computes the maximum number of redexes that can be reduced in parallel. For a tree with the root node r, the maximum number of redexes is MaxP(r, ∅, true).

6.2. Program semantics and understanding

Apart from analysing how programs can be evaluated in parallel, we can analyse semantics and flows of programs using their λ-terms. Here are some literatures discussing related topics.

First, the connection between denotational and operational semantics of a λ-calculus-like language (Programming Computable Functions; PCF) is studied in [20]. The correctness of programs, according to the specifications, could be proven using similar methods. Second, the flow of a λ-term is studied in [21], and the flow of a functional programming language 'Scheme' is studied in [22]. These two methods could be applied to programs in λ-terms to analyse the flows of the programs. Lastly, the λ-term of a program can be used in the process of automatic program understanding as studied in [23]. The knowledge base keeps programs as λ-terms in order to deduce the function of the queried program. These processes will benefit the development of AI and software engineering fields.

7. Conclusion

This paper shows how to represent programs using λ-calculus and how to evaluate λ-terms, proposes an algorithm for parallel β-reduction, and reviews some analysing methods on λ-terms. We also discuss on how simply typed λ-calculus is not Turing complete and it cannot be used to represent programs. Here, we compare pros and cons of representing programs in λ-calculus with ideas from [5].

For pros, λ-calculus is an appropriate model for representing programs in a mathematical form. The terms can be evaluated in parallel without specifying in programs. And, λ-terms look close to the specification than imperative programs. For cons, it is harder for humans to understand and debug the errors of λ-terms. Also, the processing time of λ-term evaluation is much larger, since it uses unary number system.

Using λ-calculus to represent programs may not be popular now. More methods on λ-term properties analysis are needed to be studied. Also, our algorithm needs to be analysed whether it will be efficient when considering several reduction steps. We let this be in future research.

8. References

[1] A. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*. 1937, **2** (4): 153-163.

[2] J. Hindley, J. Seldin. *Lambda-calculus and combinators: an introduction*. Cambridge University Press, 2008.

[3] G. Hillebrand, P. Kanellakis. On the expressive power of simply typed and let-polymorphic lambda calculi. In: *Logic in Computer Science (LICS'96) Proceedings*. 1996, pp. 253-263.

[4] M. Might. *Compiling to lambda-calculus: Turtles all the way down*. [Online; accessed 30-December-2016].

[5] H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*. 1997, **3** (2): 181–215.

[6] G. Lewis, G. Sussman. *Lambda: The ultimate imperative*. 1976.

[7] R. Machado. An introduction to lambda calculus and functional programming. In: *Theoretical Computer Science (WEIT), 2013 2nd Workshop-School on*. 2013, pp. 26-33.

[8] K. Dosen, Z. Petric. *The maximality of the typed lambda calculus and of cartesian closed categories*. arXiv preprint math/9911073, 1999.

[9] S. Fortune, D. Leivant, M. O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM (JACM)*. 1983, **30** (1): 151-185.

[10] O. Kiselyov. *Lambda calculus and lambda calculators: Predecessor and lists are not representable in simply typed lambda-calculus*. [Online; accessed 18-February-2017].

[11] H. Schwichtenberg. Definierbare funktionen imλ-kalkül mit typen. *Archive for Math. Logic*. 1975, **17** (3): 113-114.

[12] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*. 1979, **9** (1): 73-81.

[13] M. Zakrzewski. *Definable functions in the simply typed lambda calculus*. arXiv preprint cs/0701022, 2007.

[14] R. Kashima. *A proof of the standardization theorem in λ-Calculus*. 2000.

[15] M. Takahashi. Parallel reductions in λ-calculus. *Information and computation*. 1995, **118** (1): 120-127.

[16] H. Xi. Upper bounds for standardizations and an application. *The Journal of Symbolic Logic*. 1999, **64** (1): 291-303.

[17] G. Gonthier, M. Abadi, J. Lévy. The geometry of optimal lambda reduction. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1992, pp. 15-26.

[18] J. Lamping. An algorithm for optimal lambda calculus reduction. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 16-30.

[19] J. Lévy. Optimal reductions in the lambda calculus. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. 1980, pp. 159-191.

[20] G. Plotkin. LCF considered as a programming language. *Theoretical computer science*. 1977, **5** (3): 223-255.

[21] N. Jones. Flow analysis of lambda expressions. *Automata, Languages and Programming*. 1981, pp. 114-128.

[22] O. Shivers. Control flow analysis in Scheme. *ACM SIGPLAN Notices*. 1988, **23** (7): 164-174.

[23] S. Letovsky. Program understanding with the lambda calculus. In: *IJCAI*. 1987, pp. 512-514.