Computational Complexity via λ-Calculus Programming Paradigms Report

Patryk Tymoteusz Pilichowski

2026

1 Introduction

λ-calculus is a fundamental model of computation, equivalent in expressive power to the Turing machine yet different in its substitution-based nature. Its simplicity makes it an elegant tool for studying functional programming languages and proof theory. Defining a formal notion of computational complexity for it however, has proven nontrivial.

A core operation of λ-calculus, β-reduction, substitutes a term for a bound variable, but the computational cost of a single reduction step can vary greatly depending on the size of the substituted term and chosen reduction strategy. Simply counting the β-reductions would therefore yield an unrealistic cost model. Thus, a more sophisticated analysis and cost model is required.

The invariance thesis (introduced by Peter van Emde Boas and Cees F. Slot) defines models of computation that we call "reasonable" only if they can simulate each other with a time overhead that is polynomially bounded and a constant-factor overhead in space. Formally, if a problem can be solved in time $O(n^k)$ on one such model, then it can be solved in time $O(n^{ck})$ for some constant c on any other reasonable model.[3] If we take the Turing machine as point of reference for this, then many other models, such as RAMs and variants of the Turing machine, satisfy the invariance thesis. This report will introduce cost models that satisfy this invariance property for pure untyped λ-calculus, as well as show a process that automatically gives the complexity of a functional program.

2 Invariant cost models for λ-calculus

2.1 Weak call-by-value model

The following example would suffice to show why a raw count of β-reductions is not a good measure of computational cost. Let the term $E_n = (\lambda x.x\ x)$ be applied n times, starting on z:

$$E_1 = (\lambda x.x\ x)\ z \xrightarrow{\beta} zz$$

$$E_2 = (\lambda x.x\ x)\ E_1 = (\lambda x.x\ x)((\lambda x.x\ x)\ z) \xrightarrow{\beta} (zz)(zz)$$

$$E_3 = (\lambda x.x\ x)\ E_2 = (\lambda x.x\ x)((\lambda x.x\ x)((\lambda x.x\ x)\ z)) \xrightarrow{\beta} ((zz)(zz))((zz)(zz))$$

$$\ldots$$

Each application duplicates its argument, since x appears twice in the body. When this doubling is iterated n times, the size of the intermediate terms grows exponentially in n. Performing such substitutions naively therefore requires exponential time in total even though the raw number of β-reductions remains linear in n. It is now clear that simply counting these reductions does not provide a polynomially faithful measure of computational cost with respect to Turing machines.

To solve this problem, Dal Lago and Martini proposed the weak call-by-value cost model.[1] β-reductions are restricted in two ways here:

1. Weak: You can only reduce at the top-level or in the argument position of an application, never inside the body of a λ-abstraction.

2. Call-by-value: A β-reduction is performed only when the argument is already a value (irreducible abstraction or a variable).

Now, let's consider the already defined term $E_2 = (\lambda x.x\ x)((\lambda x.x\ x)\ z)$ and do the weak call-by-value reduction. Before reducing the outer redex, we notice that the argument is not a value yet, so we need to try to reduce it first (call-by-value): $((\lambda x.x\ x)\ z) \to z\ z$ (argument z is a value). Now we have $(\lambda x.x\ x)(z\ z)$, where the argument is an application, not a value. Therefore no more reductions are possible and the term is in its normal form.

The associated cost model, known as the difference cost model, charges each β-reduction step $M \to N$ a positive integer of $\max\{1, |N| - |M|\}$ where $|M|$ is the number of symbols in M. The total cost thus accounts for both the number of reduction steps and actual size growth from substitution. Dal Lago and Martini proved that, under this cost measure, the weak call-by-value λ-calculus and multi-tape Turing machines can simulate each other with polynomial overhead in time and constant factor overhead in space.[1]

This implies that the model satisfies the invariance thesis and thus can serve as a robust, machine-independent tool for studying complexity of, for instance, higher order functional programs.

2.2 Full λ-calculus: LO Reduction & Sharing

Many higher-order programs require the ability to reduce inside the bodies of abstractions, which the weak call-by-value model forbids. Dal Lago and Accattoli proposed a way to make full λ-calculus, with some adjustments, a reasonable model.[2] Firstly, the reduction strategy used is the leftmost-outermost (LO) strategy which always reduces the leftmost, outermost redex first.

If we stop here, the size explosion problem clearly persists. Consider the term $t_0 = u = yxx$ and $t_{n+1} = (\lambda x.\ t_n)\ u$. LO reduction reaches the normal form in n steps, but the output doubles at each step and grows exponentially in n. It is therefore evident something more is needed to be able to satisfy the invariance thesis.

The workaround to this is a method known as *sharing*, which replaces immediate copying with explicit substitutions written in the form of $t\,[x \leftarrow u]$. Rather than duplicating u everywhere x appears in one run, the copy is deferred and propagated one occurrence at a time. Now if we apply this to the same term, we produce $(yxx)[x \leftarrow u][x \leftarrow u]$ instead of the exponentially large $y(yuu)(yuu)$. This keeps the size of intermediate terms linear to n.

This is in fact a language already formalized as Linear Substitution Calculus (LSC).[4] Dal Lago's and Accattoli's contribution is proving that LSC is invariant with respect to the λ-calculus by using their defined notion of *useful* reductions and a shared normal form, with the associated cost model simply counting those useful reductions.[2] Shared normal forms are terms where every explicit substitution $t[x \leftarrow u]$ has either been applied onto t by replacing all free occurrences of x, or where x does not appear free in t at all, making the substitution "garbage-collectable".

A reduction step is *useless* if the propagation just moves an explicit substitution deeper into a term without ever reaching an occurrence of the variable it is meant to substitute. Such steps can be entirely ignored for the purposes of complexity analysis. A reduction step is *useful* if it either:

- Fires a β-redex and creates a new explicit substitution, or
- propagates an explicit substitution into a position where x actually occurs free, which makes progress toward an actual variable occurrence.

3 Complexity Analysis for Functional Programming

We now turn to the practical problem of automatically deriving concrete complexity bounds for higher-order functional programs. Pure λ-calculus cost models do not give us tools for real languages such as OCaml or Haskell which mix higher-order functions, algebraic datatypes, pattern matching, and fixpoints. Avanzini, Dal Lago, and Moser solve this complication without losing invariance by translating a higher-order program into an equivalent first-order term-rewrite system (TRS), which is a first order model designed specifically for complexity tools.[5] Subsequently, a chosen pipeline of transformations is applied, and the result is fed to existing first-order complexity provers (FOP) like TCT.[6]

A TRS consists of a finite set of function symbols and directed rules of the form $f(l_1, ..., l_k) \to r$, where evaluation repeatedly replaces matching subterms under call-by-value semantics. At each step, the system looks for a subterm that matches the left-hand side of a rule (after substituting values for variables) and replaces it by the corresponding right-hand side. For instance, the following TRS evaluates $\text{add}(S(S(0)), S(0))$ in three steps to $S(S(S(0)))$:

$$\text{add}(0, y) \to y$$
$$\text{add}(S(x), y) \to S(\text{add}(x, y))$$

Using *defunctionalisation*, every λ-abstraction and fixpoint is turned into an explicit first-order closure (e.g $Cl_1(f, g)$ for $\lambda z.f(g, z)$) and every application is routed through a single @ operator. Four complexity-reflecting transformations: inlining, dead-code elimination, instantiation (via control flow analysis) and uncurrying, simplify the system into a first order TRS. For clarity, a transformation f is complexity-reflecting if for any program A, the asymptotic complexity of A is bounded by the complexity of transformed program $f(A)$. The paper illustrates this entire process via the following list-reversal program written in OCaml:

```
let comp f g = fun z -> f (g z) ;;
let rec walk xs =
  match xs with
  | [] -> (fun z -> z)
  | x::ys -> comp (walk ys) (fun z -> x::z) ;;
let rev l = walk l [] ;;
let main l = rev l ;;
```

The process produces the following first order TRS that is fed to the TCT

for complexity analysis:

$$\begin{aligned}
\mathrm{Cl}_1(\mathrm{Cl}_2, \mathrm{Cl}_3(x), z) &\to x :: z \\
\mathrm{Cl}_1(\mathrm{Cl}_1(f, g), \mathrm{Cl}_3(x), z) &\to \mathrm{Cl}_1(f, g, x :: z) \\
\mathrm{fix_walk}([\,]) &\to \mathrm{Cl}_2 \\
\mathrm{fix_walk}(x : ys) &\to \mathrm{Cl}_1(\mathrm{fix_walk}(ys), \mathrm{Cl}_3(x)) \\
\mathrm{main}([\,]) &\to [\,] \\
\mathrm{main}(x : ys) &\to \mathrm{Cl}_1(\mathrm{fix_walk}(ys), \mathrm{Cl}_3(x), [\,])
\end{aligned}$$

4 Conclusion

The weak call-by-value model of Dal Lago and Martini shows that restricting reduction and charging for size growth makes the β-reductions provably invariant. The extension to full λ-calculus via LO reduction and sharing shows that the exponential blowup from standard substitution is a product of the representation itself, not a fundamental feature of the calculus. Thus, deferring copies through explicit substitutions makes the reductions invariant for the full λ-calculus.

By translating higher-order functional code into first-order term-rewrite systems through complexity-reflecting transformations, the framework by Avanzini, Dal Lago, and Moser makes the invariant cost models of λ-calculus usable in automated complexity analysis. The list-reversal example that mixes higher-order functions, pattern matching, and closures yields, after defunctionalisation and simplification, a TRS that existing provers can handle directly.

References

[1] Lago, U. D., & Martini, S. (2008). The weak lambda calculus as a reasonable machine. Theoretical Computer Science, 398(1-3), 32-50. https://doi.org/10.1016/j.tcs.2008.01.044

[2] Accattoli, B., & Lago, U. D. (2016). (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. Logical Methods In Computer Science, Volume 12, Issue 1. https://doi.org/10.2168/lmcs-12(1:4)2016

[3] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume A: Algorithms and Complexity, pages 1–66. North-Holland, Amsterdam, 1990

[4] Accattoli, B. (2018). Proof nets and the linear substitution calculus. In Lecture notes in computer science (pp. 37–61). https://doi.org/10.1007/978-3-030-02508-3_3

[5] Avanzini, M., Lago, U. D., & Moser, G. (2015). Analysing the complexity of functional programs: higher-order meets first-order. ACM SIGPLAN Notices, 50(9), 152–164. https://doi.org/10.1145/2858949.2784753

[6] Avanzini, M., Moser, G., & Schaper, M. (2016). TCT: Tyrolean Complexity Tool. In Lecture notes in computer science (pp. 407–423). https://doi.org/10.1007/978-3-662-49674-9_24