Internal Type-Theoretic Complexity Analysis

ASHLEY SAMUELSON, Georgia Institute of Technology

ABSTRACT

Type-theoretic frameworks for complexity analysis are important for formulating complexity-theoretic arguments, and for automating runtime analysis. Choice of framework can have a large impact on both *expressiveness*, the class of programs which the framework can reason about, and *usability*, the degree to which the framework automates the analysis, or relies on user-created proofs. In this work, we describe a framework for complexity analysis in which the runtime of language-internal terms can be evaluated and reasoned about. Our framework provides both a high level of usability by automating the calculation of exact runtimes and runtime recurrences, and a high level of expressiveness with its ability to reason about all programs expressible in the calculus of constructions. The runtime of open terms is represented by atomic language constructs, enabling analysis of higher-order functions, and creating a compositional structure. Finally, we prove a faithfulness property, showing that all internally computed runtimes for closed terms correspond to their runtimes as defined by our language semantics.

1 INTRODUCTION

Complexity analysis is the study of the runtime of programs, including exact, worst-case, and termination analysis. Type-theoretic frameworks for complexity analysis are central to the formalization of complexity-theoretic arguments, and methods for automated runtime analysis. In these frameworks, there are two important design considerations: expressiveness and usability. Expressiveness refers to the class of programs which the framework can reason about. Usability refers to the degree to which the framework automates the analysis, or relies on user-supplied proofs. The choice of framework can have a significant impact on both of these factors.

Designing a framework with both high expressiveness and usability is a difficult task, as these are opposing principles. More expressive frameworks allow for the analysis of a broader range of program classes, but require the user to manually conduct runtime analysis or provide typing derivations. On the other hand, more usable frameworks can automatically compute runtimes for any program defined within them, but severely limit the class of programs which may be reasoned about.

This work describes a type-theoretic framework for complexity analysis which has both a high level of expressiveness and usability. Our key insight into solving this problem is that when a term is well-defined in our underlying type theory, the calculus of constructions, it must have a valid typing derivation, and hence a terminating evaluation. Using this fact, our framework can automatically determine the exact runtime of any closed term, and calculate runtime recurrences for terms with free variables. This capability allows our framework to have a high level of expressiveness, as it can analyze any program written in the calculus of constructions, and a high level of usability, as runtime recurrences are computed automatically, without any need for user proof. We describe our framework as an *internal* approach to complexity analysis as it allows for reasoning about programs that are defined within the type theory itself. Our approach also naturally handles higher-order arguments and function composition, and may be modified to support different computational models or more expressive base theories.

2 RELATED WORK

There are two main approaches in existing type-theoretic complexity analysis frameworks: the *encoding-based* approach, and the *language-based* approach.

In the encoding-based approach, an encoding of a class of well-formed, terminating programs is defined, along with a function to evaluate their runtimes. See for example Constable's [3] definition of an inductive class of polynomial time functions closed under recursion and composition. Although encoding-based methods provide an automatic way to compute runtimes, this approach may limit the class of considered programs to those within a single complexity class, or those with a simple structure, limiting expressiveness.

In the language-based approach, a formal syntax and semantics for a language are defined, allowing the user to conduct complexity analysis through explicit proofs. This approach is particularly useful in performing automated runtime analysis for functional languages [2, 4]. While this framework allows for the analysis of arbitrary languages, most formalizations typically involve complex machinery, and require the user to manually create proofs of programs runtimes based on the formalized semantics, limiting usability.

3 MOTIVATING EXAMPLE

Consider the map function on lists. Intuitively, one expects that the time to map a function `f` onto a list `xs` should be the sum of the times to evaluate the application `f e` for each `e` ∈ `xs`. As our language allows for reasoning about the runtime of higher-order arguments, the following statement can be shown for all `f : A → B` and `xs : List A`. Terms surrounded by $\langle\cdot\rangle$ can be read as unevaluated/held computations, and terms surrounded by $|\cdot|$ can be read as fully evaluated/forced computations.

$$\textbf{time}\langle|\texttt{map f}|\ |\texttt{xs}|\rangle =_{\textbf{Nat}} \texttt{len xs + sum (map } \textbf{time}\langle|\texttt{f}|\ |\_|\rangle\ \texttt{xs)}$$

This follows from the definition of the atomic construct **time**, which allows for runtimes to be analyzed. The following definitional equalities hold, allowing the above to be shown by induction.

$$\begin{aligned}\textbf{time}\langle|\texttt{map f}|\ |\texttt{[]}|\rangle &\equiv 0\\ \textbf{time}\langle|\texttt{map f }|\texttt{x::xs}|\rangle &\equiv 1 + \textbf{time}\langle|\texttt{f}|\ |\texttt{x}|\rangle + \textbf{time}\langle|\texttt{map f}|\ |\texttt{xs}|\rangle\end{aligned}$$

With sufficient infrastructure, one could conduct automatic worst-case asymptotic analysis on more complex functions, such as sorting algorithms, and prove statements such as the following worst-case bound for a sorting function with a constant-time comparison function `f`.

$$\textbf{time}\langle|\texttt{sort f}|\ |\texttt{xs}|\rangle \leq\ \texttt{c * (len xs) * log (len xs)}$$

4 INTERNAL COMPLEXITY ANALYSIS

Our base type theory is the calculus of constructions extended to contain the type of natural numbers **Nat** and the propositional equality type $=_A$. We also add an atomic type family of *computations*, **Comp**(A). A term of this type represents an unevaluated computation which, when evaluated, results in a value of type A. This type has the monadic constructors **pure**, **fmap**, **liftA** and **bind**, and an evaluation function **eval** to extract the value represented by the computation. The runtime

of a computation can be evaluated by the atomic function **time**.

$$\textbf{pure} : A \to \textbf{Comp}(A) \qquad \textbf{fmap} : (\prod_{x:A} B\ x) \to \prod_{a:\textbf{Comp}(A)} \textbf{Comp}(B\ \textbf{eval}(a))$$

$$\textbf{eval} : \textbf{Comp}(A) \to A \qquad \textbf{liftA} : \textbf{Comp}(\prod_{x:A} B\ x) \to \prod_{a:\textbf{Comp}(A)} \textbf{Comp}(B\ \textbf{eval}(a))$$

$$\textbf{time} : \textbf{Comp}(A) \to \textbf{Nat} \qquad \textbf{bind} : (\prod_{x:A} \textbf{Comp}(B\ x)) \to \prod_{a:\textbf{Comp}(A)} \textbf{Comp}(B\ \textbf{eval}(a))$$

$$\frac{e \Downarrow \textbf{pure}(v)}{\textbf{time}(e) \Downarrow 0} \qquad \frac{e \Downarrow \textbf{liftA}(v_1, v_2) \quad \tau[\![\textbf{eval}(v_1)\ \textbf{eval}(v_2)]\!] \Downarrow t}{\textbf{time}(e) \Downarrow t}$$

$$\frac{e \Downarrow \textbf{fmap}(v_1, v_2) \quad \tau[\![v_1\ \textbf{eval}(v_2)]\!] \Downarrow t}{\textbf{time}(e) \Downarrow t} \qquad \frac{e \Downarrow \textbf{bind}(v_1, v_2) \quad \tau[\![\textbf{eval}(v_1\ \textbf{eval}(v_2))]\!] \Downarrow t}{\textbf{time}(e) \Downarrow t}$$

τ is a meta-function which determines the runtime of a term, where the runtime of applications with open terms is interpreted through the **time** function. Some of its values are shown below, where $v[\![\cdot]\!]$ denotes the normalized value of a term.

$$\tau[\![x]\!] = 0 \qquad \tau[\![e_1 + e_2]\!] = e_1 + \tau[\![e_1]\!] + \tau[\![e_2]\!]$$

$$\tau[\![(\lambda x : A.e_1)\ e_2]\!] = 1 + \tau[\![e_2]\!] + \tau[\![e_1[v[\![e_2]\!]/x]]\!] \qquad \tau[\![\textbf{eval}(\textbf{pure}(e))]\!] = \tau[\![e]\!]$$

$$\tau[\![x\ e]\!] = \textbf{time}(\textbf{fmap}(x, \textbf{pure}(e))) + \tau[\![e]\!] \qquad \tau[\![\textbf{eval}(\textbf{fmap}(e_1, e_2))]\!] = \tau[\![e_1\ \textbf{eval}(e_2)]\!]$$

We present four claims about this language to demonstrate its usability and soundness. The first three are preservation, strong normalization, and consistency.

Theorem 4.1. *Preservation*

$$\frac{\Gamma \vdash e : A \quad e \Downarrow v}{\Gamma \vdash v : A}$$

Theorem 4.2. *Strong Normalization*

$$\frac{\Gamma \vdash e : A}{\exists! v \in \mathit{Val}\ \ e \Downarrow v}$$

Theorem 4.3. *Consistency*

$$\frac{}{\cdot \nvdash e : \emptyset}$$

Preservation can be shown to hold via a standard argument by induction on typing derivations and the reduction relation. There are multiple [1, 5, 6] works on the strong normalization of the calculus of constructions. Although these arguments are too involved to provide a short account, they can be extended to our language as the **Comp** type may be represented by an inductive type family, and as the semantics of the **time** function are deterministic, and respect judgmental equality. Finally, consistency follows from preservation and strong normalization.

Our final claim relates to the faithfulness of the **time** function with respect to our formalized complexity semantics. Intuitively, this claim states that for any term representable by a computation, the runtime of the computation as computed by **time** corresponds with the number of steps required to fully normalize the term. Formally, for any closed computation e with normal form v, the value of $\textbf{time}(e)$ corresponds with the number of steps required to normalize $\textbf{eval}(v)$. The notation $e \Downarrow_t v$ denotes that e reduces to value v in t steps within our semantic model, and similarly $e \Downarrow_t$ denotes that e reduces to some value in t steps.

Theorem 4.4. *Faithfulness*

$$\frac{\cdot \vdash e : \textbf{\textit{Comp}}(A) \quad \textbf{\textit{time}}(e) \Downarrow t \quad e \Downarrow v}{\textbf{\textit{eval}}(v) \Downarrow_t}$$

This claim can also be shown by induction on typing derivations and the reduction relation, and relies on the correspondence between our formal semantics and the meta-function τ.

REFERENCES

[1] T. Altenkirch. Proving strong normalization of cc by modifying realizability semantics. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 3–18, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[2] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1):79 – 103, 2004. Implicit Computational Complexity.

[3] R. L. Constable. Expressing computational complexity in constructive type theory. In D. Leivant, editor, *Logic and Computational Complexity*, pages 131–144, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[4] N. Danner, D. R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. *CoRR*, abs/1506.01949, 2015.

[5] H. Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, pages 14–38, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[6] M. Stefanova and H. Geuvers. A simple model construction for the calculus of constructions. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, pages 249–264, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.