Certifying Time Complexity of Agda Programs Using Complexity Signatures

Hans Hüttel Christian Bach Møllnitz Johannes Elgaard
Simon Rannes hans@cs.aau.dk christianmoellnitz@gmail.com
joha4270@gmail.com sqrannes@gmail.com

Department of Computer Science, Aalborg University, Denmark

1 Introduction

This paper relates to the line of work that attempts to verify fundamental algorithms in computer science by means of certified implementations. Coppello showed that the merge sort algorithm is correct [1]. In a series of papers, Firsov built certified implementations of central algorithm in formal language theory in the dependently typed functional language Agda. In [3], it was shown by Firsov and Uustalu that the CYK algorithm for context-free parsing is correct.

This line of work deals with termination and correctness; performance is a lesser concern and to a certain extent is left unexamined which is arguably failing to certify one of the most important characteristics. This is especially noteworthy, since the CYK algorithm has low polynomial-time complexity which in itself shows that the class of context-free languages is a subclass of P.

All of these approaches reason about time complexity by significantly altering the code to be analysed. With [9] and [2] the cost is included in the syntax of the solution language, and with [5] a translation to a complexity specific language is performed.

In this paper we introduce an approach based on what we call complexity signatures and is inspired by Gurr [4] who shows how complexity may be viewed as a monad constructor. The `Timed` monad that we introduce allows us to not modity the implementation itself but to reason about the time complexity of an algorithm by only annotating functions. Both our implementation and the complexity analysis are carried out within Agda [6].

2 Complexity analysis of the CYK algorithm using the `Timed` monad

The goal of our approach is to embed computational cost into the code, while operating on what is recognizably the same code. We do this by taking the signature of a function and replacing every function it calls with a complexity annotated variant.

The underlying mechanism behind complexity signatures is that of changing a given value, a, to be associated with its own cost, by putting it in a pair (a, n), where n is the time it has taken to produce this value. Functions operating on those values then sum the time taken to produce its parameters and increment it by some cost, based on how expensive the function itself is.

For this, we introduce the monadic type `Timed`, representing a value a of type A produced in n time.

```
data Timed (A : Set) : Set where
  time : ℕ → A → Timed A
```

First we introduce the prime operator (`_′`). It converts a value into one produced in zero time. The implementation calls `time` and corresponds to the usual monad function `return`.

```
_′ : {A : Set} → A → Timed A
x ′ = time 0 x
```

We also define the bind operator, which in this monad increments the complexity of a given function by n.

```
_»=_ : {A B : Set} → Timed A → (A → Timed B) → Timed B
_»=_ (time n x) fn = fn x ′′′ n
```

We count computation steps by means of the `lift` function. It converts a function into a function that takes timed arguments and produces its result in the time of its argument plus one, $T_{lift} = T_A + T_B + 1$ – thereby counting the number of function calls.

```
lift : {A B C : Set} → (A → B → Timed C) → (Timed A → Timed B → Timed C)
lift = lift″ ∘ lift′
```

Our version of the CYK algorithm follows that of Valiant [8]. Given a CFG G in Chomsky Normal Form, we define a dot product on n-dimensional vectors of sets of nonterminals V_G in G and then use this to define matrix multiplication. The algorithm will, given an input string $w = w_1 \dots w_n$, form an $n \times n$ matrix A such that A_{ii} contains the set of nonterminals that can immediately derive w_i. We then compute the elements $A, A^2, A^3 \dots, A^n$. At the kth step, the entry A^k_{ij} contains precisely the set of nonterminals that derive the substring $w_i \dots w_j$. We have that $w \in L(G)$ iff A_{1n} contains the start symbol S.

In Agda we define `Element` as being a list of nonterminal symbol and define the union, _∪ _ as two `Element`s appended with duplicates removed. A function crucial to the algorithm is _•_, which calculates the product of two elements in the matrix by finding matching nonterminal symbols that match rules in the language. Its time-annotated version is defined below. Note that we split the definition in two.

```
_•′_ : {n : ℕ} → Vec ℤ n → Vec ℤ n → Timed ℤ
[] •′ [] = 0ℤ′
(x :: xs) •′ (y :: ys) = (x ′ *″ y ′) +″ (xs •′ ys)

_•″_ : {n : ℕ} → Timed (Vec ℤ n) → Timed (Vec ℤ n) → Timed ℤ
_•″_ = lift″ _•′_
```

Based on this definition of dot product, we can now define matrix multiplication in our setting. To do this, we define the union operator, and an auxiliary function `⇒_∘_` that determines which rules could be applied in a row and column. Along with our library of complexity certified matrix operations we can now define a specialized matrix multiplication _⊗_ operator that operates on matrices whose entries are sets of nonterminals. Its time-annotated version is

```
_⊗′_ : {n m j : ℕ} → (suc n) × (suc m) → (suc m) × (suc j)
  → Timed ((suc n) × (suc j))
_⊗′_ = MatrixMultiplicationBase Element _•″_
```

In our proof of the correctness of the CYK algorithm, we start by certifying the complexity of `_∪′_` with `∪′-cost-proof`. This is the used as justification to prove `•′-cost-proof`, that the complexity of the individual operations is based on the size of the elements, and therefore the total number of rules. Here, Agda can not guarantee termination and we have to insert the Terminating flag. Finally, `⊗-cost-proof`, shows that the complexity bound of it is similar to that of the matrix multiplication defined in the matrix library, with the complexity of the two nested `tabulate′`s, `matrixToRow′` and `matrixToColumn′` functions added on top of the complexity of the _•′_ operator.

```
⊗-cost-proof : {n m j : ℕ} → (m₁ : n × m) → (m₂ : m × j)
  → cost (m₁ ⊗′ m₂) ≤ n * (j * (m * (№nonterminals * 2 +
  №nonterminals * №nonterminals * (№nonterminals + №rules)) +
```

```
  n + m * suc j))
⊗-cost-proof m₁ m₂ = matrix-multiply-cost-proof _•′_
  (№nonterminals * 2 + №nonterminals * №nonterminals
  * (№nonterminals + №rules)) •′-cost-proof m₁ m₂
```

We show that our CYK implementation has a time complexity of $O(n \cdot j \cdot m \cdot (n^t \cdot 2 + n^t \cdot n^t \cdot (n^t + n^r)) + n + m \cdot suc\, j)$ which since $j = n = m$, simplifies to $O(n^4)$. Because of our simple approach to transitive closure we never reduce recognition to a single multiplication and therefore we have to perform n matrix multiplications, so the total complexity is $O(n^5)$.

3 Conclusion

We have shown how to verify the CYK algorithm. Our implementation in Agda uses a matrix library; the definition of our certified functions can be almost identical to normal functions written in Agda, and that the only mandatory alteration requirement is the need to encapsulate the return type in a `Timed` monad.

Our approach makes the code itself less convoluted. However, unlike the other approaches mentioned, every annotated function needs to have its complexity certified separately, as the proof of the complexity of a composite function need proofs of the complexity of each constituent function.

The manual nature of proving complexity means that a complexity bounds correctness relies on the programmer to annotate correctly. A proof that a function has a complexity of $\mathcal{O}(n)$ is formulated such that it will also prove that the complexity is $\mathcal{O}(n^2)$. It is possible to use the library for certifying the Θ of a complexity bound, but not always practical if the runtime varies depending on the value of input parameters, not just their size.

References

[1] E.. Copello, A. Tasistro & B. Bianchi (2014): *Case of (Quite) Painless Dependently Typed Programming: Fully Certified Merge Sort in Agda*. In F.Pereira, editor: *Proc. of SBLP 2014*, *LNCS* 8771, Springer, pp. 62–76, doi:10.1007/978-3-319-11863-5_5.

[2] N.A. Danielsson (2008): *Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures*. *SIGPLAN Not.* 43(1), p. 133–144, doi:10.1145/1328897.1328457.

[3] D. Firsov & T. Uustalu (2014): *Certified CYK parsing of context-free languages*. *J. Log. Algebr. Meth. Program.* 83(5-6), pp. 459–468, doi:10.1016/j.jlamp.2014.09.002.

[4] D. J. Gurr (1990): *Semantic frameworks for complexity*. Ph.D. thesis, University of Edinburgh, UK.

[5] B. Hudson (2016): *Computer-Checked Recurrence Extraction for Functional Programs*. Masters Thesis, Wesleyan University.

[6] U. Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Dept. of Comp. Sci. and Engineering, Chalmers University of Technology, Sweden.

[7] M. Sipser (2013): *Introduction to the Theory of Computation*. Cengage Learning.

[8] L. G. Valiant (1975): *General Context-Free Recognition in Less than Cubic Time*. *J. Comput. Syst. Sci.* 10(2), p. 308–315, doi:10.1016/S0022-0000(75)80046-8.

[9] P. Wang, D. Wang & A. Chlipala (2017): *TiML: A Functional Language for Practical Complexity Analysis with Invariants*. *Proc. ACM Program. Lang.* 1(OOPSLA), doi:10.1145/3133903.