07-300 Project Proposal: Mechanization and Automation of the Cost Aware Logical Framework by Harper et al.

Ethan Chu

https://www.andrew.cmu.edu/user/ethanchu/07300/

1 Project Description

1.1 Background

Type systems are a central part of programming languages. They formalize and enforce the categorization of the various components of a programming language, including expressions, functions, modules, and more. By specifying exactly how different parts of a program interface with each other, they help minimize the chances of bugs, facilitate compiler optimizations, and even act as a form of documentation.

Yet we claim that most programming languages do not exploit the power of type systems to the fullest extent possible. Consider sorting a list of natural numbers. In common notation, we may express the type `sort : nat list -> nat list`, indicating that `sort` is a function that takes as input a list of natural numbers, and outputs a list of natural numbers, with the implication that the output list is sorted. These type guarantees are certainly helpful, but do not go very far otherwise. There are countless other functions that could easily have the same type as `sort` as specified, such as a function that increments all elements in the list by one, `incr : nat * nat -> nat`. Another interesting concern we may have is even among all correct sorting algorithms with the specified type, they each have different cost structures. `sort` implemented via insertion sort would certainly differ in cost from `sort` implemented via merge sort.

We may be tempted to leave such matters of behavior and cost, and just accept that our types cannot convey everything. However, it is actually quite possible and interesting to consider the ramifications of a type system that handles behavior and cost of programs, i.e. at compile time. This is exactly what Harper et al.'s Cost Aware Logical Framework (CALF) does. By wielding a rich dependent type theory, CALF is able to investigate the cost and behavior of programs in the same language that the program is written in. Furthermore, CALF incorporates a modal distinction between the intensional (cost) and extensional (behavior) phases, ensuring that the incurrence of cost only has effect in intensional fragments, and does not affect the extensional behavior of the program. With regards to cost specifically, CALF is able to handle several common and effective techniques, including recurrence relations and the physicist's method for amortized analysis.

Building on our previous example, when analyzing `sort` implemented via insertion sort versus mergesort in the extensional phase, CALF will verify that they behave the same, i.e. they are both correct sorting algorithms. On the other hand, in the intensional phase, given some measure of cost, say the comparison cost model, we can prove in CALF that the cost of mergesort is bounded by $n\lceil \log_2 n \rceil$, while the cost of insertion sort is bounded by n^2, where n is the length of the list being sorted.

1.2 Mechanization

CALF, unlike some of its contemporaries, is not fully automated. This means in order to verify the behavior and cost of a program, a proof must be provided in the framework. Only then can this proof be automatically verified by the proof assistant Agda, which is what CALF is written in. For simplicity, I will refer to the process of developing a proof for a program as mechanization.

The CALF team has already mechanized several interesting algorithms. In addition to the insertion sort vs mergesort example mentioned in the previous section, their team has also mechanized a proof of tight bounds for the Euclidean algorithm for greatest common divisor, as well as the amortized complexity of batched queues.

In CMU classes like 15122, 15150, and 15210 that teach fundamental algorithms, the proofs of correctness and

cost are typically derived manually. 15122 makes an effort to prove correctness via the C0 language's builtin contract system, but otherwise, correctness and cost are verified by manual computations. When justifying the cost of recursive functions in 15210 for example, students are typically asked to derive some recurrence relation, justify it via the operation of the algorithm, and then use tools such as the brick method to justify some closed form bound for the cost.

While it may be pedagogically wiser and more convenient to stick to such manual derivations, formally mechanizing the proofs of many of these algorithms is certainly a worthy endeavor, now that CALF is available. As such, this will be the primary goal of the first phase of my research project. This would include verifying the correctness and costs of a diverse assortment of algorithms and data structures, such as but not limited to

- sets and dictionaries
  - AVL trees (122)
  - red-black trees (150)
  - splay trees (451)
- priority queues
  - binary heaps (122)
  - leftist heaps (210)
- sequences (150, 210)
  - functions: map, reduce, filter, etc.
  - implementations: ArraySequence, TreeSequence, ListSequence, etc.

One unfortunate limitation for the time being is that CALF cannot handle nondeterminism yet. This means algorithms like randomized quicksort or data structures like treaps cannot be mechanized in CALF just yet.

In this phase of the project I will primarily be mentored by Professor Bob Harper and student Harrison Grodin, both on the team that built CALF. I will likely work most closely with Grodin, since he was largely responsible for the existing mechanizations showcased in their paper, most notably the insertion vs merge sort example.

1.3 Automation

As mentioned in the previous section, CALF is not fully automated, and always requires a user supplied proof which it then verifies. This can certainly limit the appeal of CALF for many, who seek a fully automated system that can compute the cost of a program. Incidentally, one such team working on such a system is that of Professor Jan Hoffmann, who is also at CMU.

Using linear type systems, Professor Jan Hoffmann has built Resource Aware ML (RaML), which is part of a series of work in automated amortized resource analysis (AARA). As its name suggests, it uses the physicist's method for amortized analysis to automatically compute the cost of programs. Initially the automation was only able to compute costs for first-order functions, and only output polynomial costs, but in recent years, advancements have allowed analysis of higher-order functions as well as computation of exponential and logarithmic bounds.

Nevertheless, there are still shortcomings. The cost bound language is significantly less expressive and not as tightly integrated into the programming language itself. More importantly, these automated cost analyzers often fail on a few functions when analyzing a large program, and become unable to proceed with little means of assisting them.

This is where CALF could potentially come to the rescue. If the project progresses to this stage, it would certainly be worth exploring the integration of CALF and RaML, and have both supplement the others' weaknesses. Specifically, RaML's automation could handle many easier cases, and when it runs into trouble, it could clearly specify such issues to the user. At this point, the user would only have to write CALF proofs for the parts of the program that RaML could not automatically analyze, which upon verification would need to be passed back to RaML somehow to resume cost analysis.

For this phase of the project, I would be primarily mentored by Professor Jan Hoffmann to gain a deeper understanding of his work on RaML and AARA. Then I would continue to work with Professor Bob Harper and Harrison Grodin as well to work out how exactly CALF and RaML would interact.

2 Goals

2.1 75% Goal

- Become familiar with the proof assistant Agda. To this end I hope to have worked through the book *Programming Language Foundations in Agda (PLFA).*
- Become familiar enough with CALF to reproduce a proof in the CALF paper (e.g. insertion vs merge sort) from scratch.
- Mechanize all the algorithms and data structures listed in section l.2.

2.2 100% Goal

- Achieve all items under the 75% goal.
- Become familiar with RaML, and implement a translation from RaML's outputted cost to a CALF proof. Specifically, this would entail taking information from RaML's intermediate computations, and generating an Agda proof of program cost from it.

2.3 125% Goal

- Achieve all items under the 100% goal.
- Implement a way for RaML to detect where it needs help when it gets stuck, so it knows exactly what CALF proofs to query the user for.
- Implement a way for RaML to then incorporate the user provided CALF proof to resume cost analysis.

3 Milestones

3.1 1st Technical Milestone for 07-300

I believe a good milestone for the end of 07-300 would be achieving the first 2 bullet points in the 75% goal. Namely, I should have worked through *PLFA* to familiarize myself with with Agda, and become able to derive a CALF proof from scratch for insertion sort.

3.2 Bi-weekly Milestones for 07-400

3.2.1 February 1st

At this point, I should have refamiliarized myself with CALF after winter break, and mechanized AVL trees.

3.2.2 February 15th

I should have finished the mechanization of the sets and dictionaries data structures listed in section 1.2.

3.2.3 March 1st

I should have mechanized priority queues and begun mechanizing sequence operations (section 1.2).

3.2.4 March 15th

I should have finished mechanizing sequence operations, and begun mechanizing simple graph algorithms like DFS and BFS.

3.2.5 March 29th

By this point, I hope to have completely finished the mechanization phase of my project, i.e. achieved the 75% goal completely. Compared to the previous milestone, this could mean wrapping up the mechanizations not stated in section 1.2, including some harder graph algorithms like shortest-path.

I also hope to have begun learing RaML and met with Professor Hoffmann at least once more.

3.2.6 April 12th

At this point, I should be decently familiar with RaML and how it uses AARA to compute the cost of programs. I should also have started implementing the translation of RaML's cost computations to CALF, perhaps getting it to work for a simple recursive example.

3.2.7 April 26th

At the end of the semester, the goal would be to have achieved the 100% goal, i.e. additionally devised a translation from RaML cost to a CALF proof on top of the mechanizations.

4 Literature Search

Of course, the most important paper for me to carefully read and understand is the CALF paper itself [5]. I am also certain that in the process of understanding the CALF paper, I will have to consult the papers that it builds on, including Kavvos's paper [4] on automatically extracting cost recurrence relations and Sterling's paper [6] which contributed to phase distinctions used in the CALF paper.

As I move into the automation phase of the project, I would also have to read the RaML and other AARA papers in greater detail. This would include Hoffmann's various papers on RaML [1] [2], as well as the work of other groups on AARA, such as the team that resolved analyzing logarithmic costs [3].

5 Resources Needed

Considering that CALF is built on the Agda Proof Assistant, it will be the primary piece of software that I will use throughout the project. I have already installed Agda and associated IDE tools, as well as the aforementioned *Programming Language Foundations in Agda*.

References

[1] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *International Conference on Computer Aided Verification*, pages 781–786. Springer, 2012.

[2] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 359–373, 2017.

[3] Martin Hofmann, Lorenz Leutgeb, Georg Moser, David Obwaller, and Florian Zuleger. Type-based analysis of logarithmic amortised complexity. *arXiv preprint arXiv:2101.12029*, 2021.

[4] GA Kavvos, Edward Morehouse, Daniel R Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[5] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *arXiv preprint arXiv:2107.04663*, 2021.

[6] Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM (JACM)*, 68(6):1–47, 2021.