CHUMP: A Mechanically Verified Implementation of Curricle

CMSC 838L — Final Project Report

JUSTIN FRANK, University of Maryland, USA
PIERCE DARRAGH, University of Maryland, USA

ACM Reference Format:
Justin Frank and Pierce Darragh. 2024. CHUMP: A Mechanically Verified Implementation of Curricle: CMSC 838L — Final Project Report. In *Proceedings of Proceedings of the First Occurrence of CMSC 838L (838L).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Curricle is an existing work that resolves some constraints in the implementation of systems for batteryless energy-harvesting devices. The approach develops a new type system that encodes a first-class notion of idempotence that, when combined with inferred information about input-taintedness and memory usage patterns, can guarantee a notion of safety for these systems by ensuring certain variables are checkpointed at appropriate points. The authors argue that this type-system-based approach strikes a balance between ease of use for programmers and reduced checkpointing overhead.

However, the theoretical underpinnings of Curricle are incredibly complex, consuming nearly 50 pages of semantic diagrams and proofs in the extended form of the publication. Believing in the claims of the authors but wanting to lend credibility to the theory of the system, we seek to mechanically verify Curricle in Rocq (formerly Coq). Unfortunately, this report details an as-yet-incomplete attempt at accomplishing this goal. This is due in large part to the fundamental difficulty of mechanical verification, but it may also be the case that the authors' choices early in the project development negatively impacted the long-term outcomes. We investigate these shortcomings and reflect on the current status and potential future of this work.

1 INTRODUCTION

Curricle [3] attempts to solve a common problem in the design space of programs for batteryless energy-harvesting devices (EHDs). EHDs have the property that they might turn off at any time during execution, which normally leads to the loss of necessary information in the course of computing something. To solve this, EHDs typically feature both volatile and non-volatile memory that programmers can use to save information across power failures. EHD systems are deployed with mechanisms for restoring the last-active memory state so they can resume execution.

One of the biggest hurdles in the use of such systems is the burden placed on programmers to correctly maintain their invariants. Various infrastructures have been developed in service of this goal, one category of which is *checkpointing*. In a checkpointing system, variables are marked for saving at specific points, meaning their information will be stored in non-volatile memory. Upon restoring from a power failure, the EHD will resume the program and the checkpointing system will restore the values of the checkpointed variables, resuming its execution from the last completed checkpoint.

Authors' addresses: Justin Frank, University of Maryland, College Park, MD, USA; Pierce Darragh, University of Maryland, College Park, MD, USA.

838L, Spring 2024, College Park, Maryland, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, the designers of the checkpointing systems have to balance ease of use with performance. The system could be made to automatically save all variables at each checkpoint, such that the programmer need only specify the checkpoint locations, but this incurs significant performance overhead in terms of both speed and power consumption. On the other hand, the system could leave the choice of which variables to checkpoint to the programmer and do nothing automatically, but this places an enormous burden on the programmer and is prone to failure (e.g., the programmer may forget to checkpoint an important variable).

Curricle provides a type-system-based approach to resolving this problem. Instead of requiring the programmer to checkpoint variables manually, they need only specify which variables are non-idempotent and set the checkpoints. From there, Curricle checks that the program is consistent with respect to the idempotence restrictions and checkpoint locations. The end result is a program that does not checkpoint more than needed while reducing the burden on the programmer. In addition, moving these concerns into the type system means programs that are not consistent will be rejected up-front, signaling to the programmer what they need to correct.

We found Curricle to be an impressive development, but a look through the full technical report reveals the system to be incredibly complex. Complex systems are often difficult to re-implement and can also contain subtle bugs. In service of improving the reliability of Curricle, we set out to mechanically verify the system using the Rocq theorem prover. However, because we found ourselves subjected to external time constraints and due to our lack of experience in mechanical verification, we removed some complexity from the type system to make the problem more tractable. Despite these simplifications, we experienced no shortage of setbacks and were unable to meet our goals.

In this report, we:

- Explain the simplified system.
- Discuss challenges that prohibited successful development of the system.
- Describe the character of the proofs we wish to develop.
- Evaluate the current implementation.
- Make a plan for future work.

2 CHUMP

The system supported by Curricle is built on top of Rust. This is useful to programmers because it makes the system easier to use, since it does not require a particularly strange toolchain. However, Rust features a complex memory use analysis as part of its type system. While this is a significant boon in favor of Rust's adoption as a low-level systems language (and a bonus for Curricle), it significantly complicates the development of proofs — mechanical or otherwise — that use it.

The extended Curricle report (see 3.1) contains dozens of pages of semantics and proofs, and a not-insignificant portion of these is due, at least in part, to the underlying complexity of Rust's type system. The Curricle authors' choice to include this in their formalization is reasonable given the intended use case, but the complexity of the proofs was a significant cause for concern for these authors. We set about developing a new, smaller language primarily intended to reduce the complexity of the proofs we would later develop.

2.1 Syntax

Figure 1 shows the abstract syntax of CHUMP, our significantly reduced language. In contrast to the language supported by Curricle, CHUMP does away with functions, structs, and pointer expressions of any kind. We also generalize loops, providing an indexed `break` command to terminate execution (which

$$
\begin{array}{lcl}
\langle c \rangle & ::= & \text{let } \langle x \rangle := \langle e \rangle \text{ in } \langle c \rangle \\
& | & \text{let } \langle x \rangle := \text{INPUT in } \langle c \rangle \\
& | & \text{if } \langle e \rangle\ \langle c \rangle\ \langle c \rangle \\
& | & \text{noop} \\
& | & \text{loop } \langle c \rangle \\
& | & \text{break } \langle n \rangle \\
& | & \langle c \rangle \text{ ; } \langle c \rangle \\
& | & \text{checkpoint}
\end{array}
\qquad
\begin{array}{lcl}
\langle e \rangle & ::= & \langle n \rangle \mid \langle b \rangle \\
& | & \langle e \rangle\ \langle op2 \rangle\ \langle e \rangle \\
\langle op2 \rangle & ::= & + \mid - \mid \times \\
\langle v \rangle & ::= & \langle n \rangle \mid \langle b \rangle \\
\langle x \rangle & ::= & \langle var \rangle
\end{array}
$$

Fig. 1. The abstract syntax of CHUMP.

```
Definition env        := list loc.
Record     store      := mkStore { st_next : positive; st_map : pmap loc }.
Inductive  kont       := kMt | kSeq (c2 : cmd) (E : env) (k : kont).
Definition checkpoint := kont * list (loc * val).
Definition CESKP      := cmd * env * store * kont * checkpoint.
```

Fig. 2. The definitions of the E, S, K, and P of our CESKP machine.

can be nested within conditional `if` forms to simulate `while`-loop behavior). Our hope in cutting out so many features was to greatly simplify the mechanization of the system.

2.2 Semantic Model

To model the semantics, we use a derivation of the CESK style. The choice of a CESK-based semantics was primarily due to the authors being familiar with this style, as well as a belief that such a style would be easier to mechanize compared to others. A CESK-style abstract machine features four components:

1. **C**ontrol — What to do next.
2. **E**nvironment — A map of variables to addresses.
3. **S**tore — A map of addresses to values.
4. **K**ontinuation — A context tracking what to do later.

In addition to these traditional elements, we add a fifth:

5. Check**P**oint — A safe state to restore to after reboot.

This gives us a CESK**P** machine. Figure 2 shows the Rocq definitions of the E, S, K, and P components.

In our model, we have only two varieties of continuation: the empty continuation, denoted `kMt`, and the sequencing continuation, `kSeq`. The `kSeq` continuation saves a second command, an environment, and another continuation. The second command is just what needs to be done after the first command in the sequence is fully executed. The environment is saved at the point at which the sequence is invoked, i.e., it contains only the variables that are visible to *both* commands in the sequence. Lastly, the nested continuation is needed to model complex control flow.

A checkpoint is a pair consisting of a continuation and a list of location–value pairs. The continuation represents what was left to be done in the program at the point at which the checkpoint was invoked, while the location–value pairs constitute a partial store to be restored upon reboot. These are the values of the checkpointed variables as they existed when the checkpoint was passed. As noted previously, continuations in our model contain environments, so the continuation with this partial store are enough information to restore the state of the machine as expected.

Fig. 3. The definitions of the IO events and log.

```
Variant event     := ePure       (M : CESKP)
                   | eInput      (f : Z -> CESKP)
                   | eOutput     (M : CESKP) (o : Z)
                   | eCheckpoint (M : CESKP).

Variant io_event  := io_in  (i : Z)
                   | io_out (i : Z)
                   | io_check
                   | io_reset.

Definition io_log := list io_event.

Definition state  := CESKP * io_log.
```

2.3 Input and Output

We chose to separate the notions of input and output from the CESKP abstract machine. This is a deviation from Curricle: Curricle's authors chose to consider "output" to be "the observable state of memory at the end of execution". We instead develop a distinct IO event log. This decision was arrived at after some consideration of what might be easiest to mechanically verify. Separating IO from the base abstract machine means we can develop a theory of the abstract machine on its own, and later encapsulate it within a broader context including the IO formalization. Figure 3 shows the definitions.

2.4 Simplified Checkpointing

Curricle supports multiple notions of checkpointing. Specifically, the authors support both JIT- and atomic-region-based views on checkpoint semantics, and the system is agnostic to the specific checkpoint implementation (e.g., undo vs. redo checkpointing). These axes of choice provide great flexibility for real-world systems, especially for a domain as varied as intermittent computing that employs devices and systems of all kinds.

However, complexity is the enemy of mechanized verification, so we stripped it down significantly.

Our semantics are built to handle only undo checkpointing, and we implicitly define all code as existing within atomic regions between checkpoints. Narrowing down the options in this manner gives a much smaller system, and hopefully simplifies the proof of the system later on.

3 CHALLENGES ENCOUNTERED

In the process of implementing this work, we experienced a number of challenges, many of which we have unfortunately not yet overcome.

3.1 Lack of Deep Comprehension

When we began our effort to mechanically verify Curricle, we felt that we had a reasonable understanding of the system and how we would go about proving its correctness, and we thought that we could make significant headway with what we knew then. Yet as we got into the minutiae of the actual implementation, we found that our comprehension was actually quite shallow, and this proved to be a significant setback in accomplishing much of anything.

```
Class MeetSemiLattice (A : Type) :=
  { meet : A -> A -> A

  ; meet_commutative : forall x y,   meet x y          = meet y x
  ; meet_associative : forall x y z, meet (meet x y) z = meet x (meet y z)
  ; meet_idempotent  : forall x,     meet x x          = x }.

Class MeetOrder {A : Set} `(@MeetSemiLattice A) :=
  { pre              :  relation A
  ; preorder         :: PreOrder pre
  ; partial_order    :: PartialOrder eq pre
  ; meet_consistent  :  forall x y, pre x y <-> x = (meet x y) }.
```

Fig. 4. Class definitions for semi-lattices.

We discovered the existence of an extended version of the Curricle paper, complete with many pages of semantics and proofs to explain the system in depth (which has been mentioned elsewhere, but its discovery originated here). Reading this extended paper allowed us to fill in the gaps in our knowledge, but unfortunately it filled in the gaps *too* well; the extended paper is very long, and the formalism it describes is incredibly complex. When we began this project knowing only the abridged form of the system, we felt confident that we could verify a sizable portion of it. After discovering the depth of the complete specification, however, that goal seemed to get further from our grasp.

To amend this discrepancy, we simplified our target system as much as possible while hopefully still retaining enough of the parts that make Curricle interesting, with the eventual goal of adding complexity back (2). However, as of this writing, this reduced system remains unimplemented.

3.2 Lattices

Curricle's type system features a number of lattices; namely, the access qualifiers, idempotence qualifiers, and taintedness qualifiers are all described by the paper (whether in prose or by use of symbology such $\prec$, $\preceq$, $\curlywedge$, $\sqsubset$, $\sqsubseteq$, $\sqcup$, etc) as being lattices built from pre-order or partial order relations.

To that end, these authors sought to formalize these structures by way of a generalized notion of lattices in Rocq. Our thinking was that, since so many lattices are used (i.e., more than 2), and since the proofs of the interesting parts of Curricle will need to explicitly refer to these relations, we would develop a general theory for them. This would enable reuse of proof techniques throughout our own proofs, but it would also reduce cognitive overhead.

Unfortunately, it seems that Rocq has managed to completely forego a standard implementation of lattice proofs or classes, and, indeed, these authors were unable to locate a correct implementation of such structures *at all*. We found a paper describing a tactic to solve lattices [2], and what was seemingly an implementation of this paper [4]. We spent a while attempting to adapt this code to our needs. For example, the code implements custom classes for partial orders (among other things), but such classes are defined in the Rocq standard library, so we sought to adapt a new lattice definition based on those found in this repository.

After much fiddling, we decided to develop our own (minimal) lattice implementation from scratch, derived from what we had found. We decided to only implement a semi-lattice (i.e., a lattice that only relates in one direction) and reorient the Curricle structures to fit within this definition. Our class definitions are shown in Figure 4. After this, we worked on implementing instances of this class for the Curricle structures we needed.

Alas, for another obstacle then reared its head. Consider the definition of "subtyping" for idempotence:

$$\frac{qAcc \preceq_a qAcc'}{\mathrm{Id}qAcc \sqsubseteq \mathrm{Id}qAcc'}$$

This seems simple enough: one idempotence qualifier is "less than or equal to" another if the first's constituent access qualifier is "less than or equal to" the second's. The access qualifiers are defined with the relations $\preceq_a$ and $\sim_a$:

$$\mathrm{Ck} \preceq_a \mathrm{Wt} \sim_a \mathrm{Wt}^t \oplus \mathrm{Rd} \preceq_a \mathrm{Wt} \oplus \mathrm{Rd}$$
$$\mathrm{Ck} \preceq_a \mathrm{Rd} \sim_a \mathrm{Wt} \oplus \mathrm{Rd}^t \sim_a \mathrm{Wt}^t \oplus \mathrm{Rd}^t \preceq_a \mathrm{Wt} \oplus \mathrm{Rd}$$

The operator $\preceq_a$ denotes a transitive and reflexive relation comparing access qualifiers in the fashion typical of a partial order, and partial orders are often one of the foundational pieces of a lattice definition. Meanwhile, the operator $\sim_a$ represents a notion of "equivalence" among some of the access qualifiers. These cannot all be fit into the same lattice, because they violate some of the laws of how lattices operate — namely, the presence of an equivalence means that the lattice operations cannot be commutative, and the partial order cannot be consistently reflexive.

To solve this problem, we have decided to simplify access qualifiers in our mechanization. Our $\preceq_a$ relation is defined as:

$$\mathrm{Ck} \preceq_a \mathrm{Wt}$$
$$\mathrm{Ck} \preceq_a \mathrm{Rd}$$

In other words, we have removed the $\oplus$-tagged qualifiers. This allows for viewing the relation through the perspective of a semi-lattice, and it also means we can move forward with the construction of proofs in terms of these structures.

3.3 Pointers

Our original specification for CHUMP included pointer values, along with reference and dereference operators. We spent quite a bit of time mechanizing this version of the project, including completion of proofs of progress and preservation of the base type system. Unfortunately, as we moved from that point to attempting to integrate additional Curricle systems, it became clear that pointers and notions of ownership à la Rust really do make things hard when it comes to mechanized verification.

To simplify our system (and hopefully render it tractable within the scope of a class project), we eventually threw out pointers and reference/dereference operations. We decided that if we can successfully prove the simpler system correct, these would be among the first features we would add back in.

4 PROOF SET-UP

Similar to Curricle, we develop a theory of correctness derived from the notion of *non-interference*. We say that an intermittent execution of a program is "correct" if there exists a continuous execution of the same program that, given the same inputs, arrives at the same machine state. The word "same" is doing a lot of work here, so we elaborate on what that means.

Intermittent execution is characterized by the presence of unpredictable power failures followed by reboots. If we imagine a temperature-reading system deployed in an environment with varying temperatures, it stands to reason that a temporary shut-down would have a significant impact on the temperatures read by the device relative to a similar nearby device that never experiences a shut-down. If the program state is derived from inputs from the temperature sensors, these devices would be likely to have noticeably different states at the end of a given period of time.

However, in the realm of proof, we need not actually deploy devices. Instead, we ask: "If a device with continuous execution were given the same inputs *as the last successful regions of the intermittent computation*, would the outcomes be the same?" To this end, we track the input and output as events in a log indexed by the checkpoints and power failures. Thus, after an intermittent execution, we can reconstitute a simulation of a continuous execution by discarding inputs and outputs from repeated regions of code (i.e., code that ran but did not reach a checkpoint prior to a power failure).

Unfortunately, we did not manage to specify our proofs in Rocq. This is due to the system not being fully implemented (3, 5). It seemed not worth the effort of writing down the proof statements if we did not yet have the underlying system built, since the proof statements must be written in terms of that system.

5 EVALUATION

When we began this work, we set evaluation goals we believed to be reasonable [1]. We now consider whether we have met these goals.

A full specification of the syntax and semantics of CHUMP. Although the syntax has been solidified and much of the semantics defined, it is not completely encoded in Rocq. We began this project by working on this and the (revised) Curricle type system at the same time, but as the end of the semester approached we focused more and more on the Curricle types rather than the base system. We feel that this was the right choice, considering the implementation of the Curricle system is the express purpose of the project, though it is disappointing not to have completed this.

Basic Curricle type system. We have made significant headway relative to where we began, including an implementation of the underlying type system *beneath* Curricle, but we have experienced setbacks at seemingly every possible opportunity in the implementation of the more complex features (3). The (re-)implementation of a complex system in a theorem-proving language is not straightforward, as it turns out, and we severely underestimated the effort it would take to accomplish our goals.

Proofs of correctness started, if not proven. Although we have produced verified proofs of progress and preservation for the base type system, these were not the proofs we had been particularly interested in writing. In fact, we have not even formally specified the intended proof statements in Rocq, which was a lesser form of this goal we had previously set. But this is not without reason. Without the completed Rocq implementation of the Curricle system, it is nearly impossible to actually phrase the proofs to any significant degree. Anything we write is no better than a prose statement of intent, and we do not see much point in spending effort on this while the base system remains incomplete.

Further goals. In addition to our base goals, we had set a number of additional goals we wanted to reach. Considering the state of everything else, these authors feel it not worth expanding on the further aims that we also did not satisfy.

6 FUTURE WORK

The most pressing issue for this work is the successful implementation of the basic Curricle type system. We believe that if this were to be completed successfully and proofs of correctness satisfied, the additional features would be significantly more straightforward to implement. In other words, CHUMP is very front-loaded in terms of programmer effort, and while we do not believe the rest of the project would necessarily be "easy", it seems reasonable to expect it would be a much quicker process.

6.1 Conclusion

Regretfully, we must admit defeat. The CHUMP system remains incomplete and our desired proofs unproven. However, we feel that we have learned a lot through this endeavor. Some brief takeaways:

- Mechanized formal verification is hard.
- Deep understanding of the entire system beforehand is necessary.
- Papers often do not specify things in the way we might like.
- Mechanized formal verification is hard.

REFERENCES

[1] Frank, J. and Darragh, P. 2024. Project milestone 2. *Proceedings of the first occurrence of CMSC 838L* (2024).

[2] James, D.W.H. and Hinze, R. 2009. A reflection-based proof tactic for lattices in coq. *Proceedings of the tenth symposium on trends in functional programming, TFP 2009, komárno, slovakia, june 2-4, 2009* (2009), 97–112.

[3] Surbatovich, M. et al. 2023. A type system for safe intermittent computing. *Proc. ACM Program. Lang.* 7, PLDI (Jun. 2023).

[4] Wiegley, J. 2023. A reflection-based proof tactic for lattices in coq. https://github.com/jwiegley/coq-lattice; GitHub.

[1] Frank, J. and Darragh, P. 2024. Project milestone 2. *Proceedings of the first occurrence of CMSC 838L* (2024).

[2] James, D.W.H. and Hinze, R. 2009. A reflection-based proof tactic for lattices in coq. *Proceedings of the tenth symposium on trends in functional programming, TFP 2009, komárno, slovakia, june 2-4, 2009* (2009), 97–112.

[3] Surbatovich, M. et al. 2023. A type system for safe intermittent computing. *Proc. ACM Program. Lang.* 7, PLDI (Jun. 2023).

[4] Wiegley, J. 2023. A reflection-based proof tactic for lattices in coq. https://github.com/jwiegley/coq-lattice; GitHub.