CHUMP

Final Project Report

Justin Frank Pierce Darragh

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 future of this work.

Introduction

Curricle [1] 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.

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

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

Figure 1: The abstract syntax of CHUMP.

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.

In this report, we:

- Explain the restricted system we chose to implement in lieu of the complete Curricle specification.
- Describe the character of the proofs we wish to develop for this system.
- Discuss the challenges that prohibited the successful development of these proofs, including:
  - Issues present in or deriving from the original work.
  - Issues derived from our choice of restricted system.
- Investigate the current status of the implementation.
- Make a plan for continuing this work.

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, it significantly complicates the development of proofs (mechanical or otherwise) that use it.

The extended Curricle report 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.

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 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.

Semantics

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.

```
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.
```

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

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.

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.

Figure 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.
```

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).

TODO

Challenges Encountered

In the process of implementing this work, we experienced a number of challenges, many of which we have not yet overcome. We group our challenges in two categories: those due to issues present in the original work or our comprehension thereof, and those due to our own choices.

Curiosities of Curricle

As noted previously, the design of the Curricle system is very complex due to the authors catering to a smörgåsbord of considerations. However, despite simplifying the system greatly, we found some parts very difficult to adapt.

In the first place, the Curricle paper regularly uses syntax for various algebraic structures, such as partial orders, pre-orders, and lattices, in addition to symbology traditionally used in the description of type systems. However, these authors found the notations used to be somewhat deceptive.

For example, the paper defines the following relation:

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

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 paper text accompanying the use of $\preceq_a$ describes the access qualifier as belonging to a lattice, but they are defined as a partial order:

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

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

We decided to simplify access qualifiers in our mechanization, though, so we have instead:

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

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

References

[1] Milijana Surbatovich, Naomi Spargo, Limin Jia and Brandon Lucia 2023. A Type System for Safe Intermittent Computing. *Proc. ACM Program. Lang.* 7, PLDI (Jun. 2023). DOI:https://doi.org/10.1145/3591250.