Beginning of page 2: "we show", then "give confidence". I am left confused as

to whether the CoreChkC --> CoreC compilation is shown correct (theorem with

proof), or carefully debugged and tested in plt-redex (which is fine too, I

just find the choice of terminology confusing)

It is a typo, should be "validate through randomized testing".

My interpretation of section IV.C is that the authors have a pen-and-paper

proof of Theorem 4, and that random testing using the Redex models has been

used to gain confidence in such a proof. Is this correct?

Yes, this is correct.

While reading the semantics, I found the fact that S-Def and S-DefNull are

applicable non-deterministically if n is 0 a bit confusing. Only when

reading the meta-theory section I realized that this is not a concrete issue

because well-formed heaps are such that $\mathcal{H}(0)$ is never defined. It

might be worth pointing this out early on.

Yes, you are right. $\mathcal{H}(0)$ is reserved (it is defined as Null, but malloc function never generates 0.

Furthermore, inspecting the code also suggests that the expression type (line 155) does not contain constructors for function calls (and I don't see a way to define functions either), conditionals, or strlen, and doesn't distinguish between the two forms of casting. All this contradicts figure 2, and should be clarified

It is in the CheckedC.v file, 392. <https://github.com/plum-umd/checkedc-atc/tree/master/csf22/coq>

This raises a fair amount of questions regarding the treatment of the

NULL pointer at this stage of the paper... is it modeled as 0, as returned by

`malloc`? are dynamic checks inserted by CheckedC to guarantee that no NULL

pointer is dereferenced?

dynamic checks are inserted by the Checked-C compiler to guarantee that no NULL pointer is dereferenced. However, that should be a problem of compiler not the semantics. The Checked-C semantics says that if the pointer value is 0 (a null) then the transition goes to NULL state. It is modeled as 0, and the Checked-C malloc operation never returns 0. The checked-C compiler also inserts a dynamic check to gurantee it.

The special case raises questions, e.g. why is this syntax-driven and

not type-driven?

This describes the semantic transition rules. We are using context evaluation framework to define the transition rules as the $E$ definition in Fig.3. like $\frac{x \Rightarrow y}{x+z \Rightarrow y + z}$, I don't know how type-driven can help us define transltion rules.

Fig4: it was hard to tell which cases were stuck states, or could reduce owing to a rule that was not shown

\Fig4: can the rules be presented in the same order they were introduced in

the paper?

Fixed. Please see the revision.

Fig4: S-FUN: $\vec\tau\_a$ seems unused; why?

See the revised text for explaination.

On page 6, section IIIC, paragraph "Pointer Access" mentions that checked pointers cannot be dereferenced in unchecked blocks - this looks funny, shouldn't it be the other way around? The Coq code contains the hypothesis m'=Unchecked -> m Unchecked in various rules of definition well-typed (BoundCheckedC, line 669; rule TyDeref in the code seems closest to the figure's T-DefArr and T-Def in the appendix, although it's a bit concerning that there's no 1-to-1 correspondence of the rules in the code and the paper).

Yeah. It is another typo. It should be the other way around.

Text says $u < c$ but this seems to contradict the paragraph right

after, because if $m = u$ then $u <= c$ and it seems like T-DefArr \*does\*

allow me to dereference a checked pointer in unchecked mode...? Or am I missing something?

See the revised text. The other way around. We allow to use a checked pointer in an unchecked block but not the opposite.

"Bounds Widening": can the join be always computed? this may be a hard problem...?

See the revised text. The join can be always computed in \CoreChkC because we assume the syntax of array bounds to be as simple as constants and variables. The \checkedc implementation uses SMT solver to compute the join operation, which is a future formalization direction of \CoreChkC.

Theorem 1 refers to a program $e$ being well-formed. Unless I've missed something, I didn't see such a definition in the paper.

This is a typo. forgot to add.

There appears to be a slight discrepancy between the blame theorem in Coq and the one in the paper: the paper mentions some e', which I believe should be r. Also, the Coq code has a further disjunct m=Unchecked in the conclusion.

It is a typo. It is either initially the mode is unchecked, and the program stucks without any computation, or it is in the middle of the computation and the stuck place is in an unchecked block.

IV: ghost variables in other contexts (e.g. Why3, Dafny) are used for things

that do not exist at run-time, but this doesn't seem to be the case here.

See the revised text.

In Dafny and Liquid Haskell, they refer to variables that are relevant during typechecking but irrelevant at runtime. In our context, we use ghost variables to refer to variables that are not directly visible to the programmer and managed internally by the CheckedC runtime. Is there a better name for it? Are there examples of runtime relevant ghost variables that we can cite?

I'd like to have seen a bit more motivation for using PLT redex

what aspects made the use of this tool preferable to formulating the compilation in Coq and using Quickchick to do random testing of the simulation result.

1. Redex is highly optimized for specifying judgments that are algorithmic. By writing down a typing relation, we can immediately obtain a typechecker that is executable. Same applies for the small-step evaluation relation. Translating the relations into functions in Coq is definitely doable but time-consuming, especially since compilation is embedded as part of the typing rules. It is also hard to see whether the function we define really corresponds to the relation unless we formally prove it. This issue is particularly relevant at the early stage of the development when the compilation rules were buggy and the simulation property was violated often as we added new generator cases.

In Redex, we don't have any formal guarantee either, but at least we can more easily see the correspondence because Redex is able to convert the relation into an executable version so we can specify the relation literally. This feature of Redex helped us speed up our development significantly when our compilation rules were constantly changing.

Even the extended model does not support full CheckedC, and the formalizations do not directly connect to the actual implementation. Could you discuss what the remaining delta is in terms of language features, and whether it might be feasible to grow the current formalization to one that integrates with the CheckedC code for LLVM, at least for existing Coq-formalizations of LLVM such as Vellum?

See the revised text for the difference between Coq, PLT, and LLVM implementation.

Fig 8, line 1: did you mean `bounds` instead of `count`? I'm confused

Fig 8, line 8: why null-check `p` again?

Yes, it was a typo. We'll fix it in the text.

There was originally some text explaining why we had the redundant check but was deleted for space reasons.

The first null-check comes from `strlen(p)` on line 4. The second null check comes from the pointer arithmetic `(p+1)` on line 5.

Technically, the null check at line 8 can be optimized away, but this kind of optimization would make our compilation more complex

and make our future endeavor at formally proving Theorem 4 (Simulation) more challenging.

Fig 9: if this is actual C code, then your null-check at line 6 will be

eliminated by the compiler. At line 3, you performed a pointer addition, which

is only defined when `p` is non-null. So, either `p` is non-null, and the

NULL-check can be eliminated; or, `p` is NULL, but line 3 was undefined

behavior, meaning the compiler is allowed to do anything, notably eliminate

the NULL-check. This is where I am super confused, and either:

- CoreC is not really the C language, and has different semantics...? but is

this well-defined in the context of LLVM?

- there is a problem that was not caught by the PLT-Redex-based testing.

Yes, we should clarify that CoreC is really an untyped variant of the CoreChkC model.

What we are really doing here is to present CoreC in a syntax that is identical to C to clarify the compilation process as otherwise the code would be extremely unreadable without any type information.

The semantics of CoreC is very lenient and there is no undefinedness for pointer arithmetic as everything is an integer. This is sufficient for proving that our semantics of CoreChKC does not require fat pointers, but extra care must be taken if we were actually writing a compiler from CheckedC to C. For example, to ensure the addition at line 3 is defined, we must cast p to uintptr\\_t before performing the addition.