A Review on "Conditional Contextual Refinement"

Presented by Gijung Im

Yonsei University

FCAI Lab Seminar
May 16, 2025

About the Paper

- **Title:** Conditional Contextual Refinement
- **Authors:**
  - **SNU:** Youngju Song , Minki Cho, Dongjae Lee, Chung-Kil Hur
  - **MPI:** Michael Sammler, Derek Dreyer
- **Conference:** POPL 2023

Table of Contents

Program Verification

- **Implementation**: A program we wish to verify.
- **Specification**: A property we wish to show the program satisfies.
- **Precondition**: What should be true before the code runs?
- **Postcondition**: What is guaranteed to be true after the code runs?

Verification Method

Based on	Form of Specification	Aspect of Composition
Separation Logic	$\{P\}\,C\,\{Q\}$ $\left\{ \begin{array}{l} P\text{: precondtion,} \\ C\text{: code,} \\ Q\text{: postcondtion.} \end{array} \right.$	modular reasoning about shared state
Refinement	$I \sqsubseteq S$ $\left\{ \begin{array}{l} I\text{: implementation,} \\ S\text{: specification.} \end{array} \right.$	transitive proof composition

Table 1: Based on Separation Logic vs. Based on Refinement.

Summary.

- ***Separation logic*** supports *conditional specifications* and *modular reasoning about shared state.*
- ***Refinement*** supports *programs as specifications* and *transitive composition of proofs.*

Question: Can we marry the complementary benefits of refinement and separation logic in one framework?

Marrying Separation Logic and Refinement

There are some attempts:

1. Relational Separation Logic.

- ***Simuliris*** is its implementation.
- There are two judgments: $\{P\}\ I \lesssim S\ \{Q\}$ and $I \sqsubseteq_{\text{ctx}} S$.

    - ***Relational Separation Logic*** ($\{P\}\ I \lesssim S\ \{Q\}$):
      A simulation $I \lesssim S$ with pre/post-conditions P, Q.
      It does not enjoy transitive composability.

    - ***Contextual Refinement*** ($I \sqsubseteq_{\text{ctx}} S$):
      It is implied by $\{P\}\ I \lesssim S\ \{Q\}$ for certain restricted choices of P and Q.
      It means that I refines S when placed in an arbitrary program context.
      However, $I \sqsubseteq_{\text{ctx}} S$ is un-conditional.

- $\therefore$ The benefits are kept *separate*!

2. Conditional Contextual Refinement. The first verification system to not only combine refinement and separation logic in a single framework but also *fuse* their complementary benefits together in a unified mechanism.

A Key-Value Storage

```
1 private data := NULL
2 def init(sz: int) =
3     data := calloc(sz)
4 def get(k: int): int =
5     return *(data + k)
6 def set(k: int, v: int) =
7     *(data + k) := v
8 def set_by_user(k: int) =
9     set(k, input())
```

Figure 1: Module I_{Map}.

```
1 private map := (fun k => 0)
2 def init(sz: int) =
3     skip
4 def get(k: int): int =
5     return map[k]
6 def set(k: int, v: int) =
7     map := map[k<-v]
8 def set_by_user(k: int) =
9     set(k, input())
```

Figure 2: Module A_{Map}.

- Unfortunately,

$$I_{\mathrm{Map}} \not\sqsubseteq A_{\mathrm{Map}}.$$

- It only holds *conditionally*:

    init should be called at most once and before any other operation.

A Key-Value Storage

$$\forall sz. \{\lfloor pending \rfloor\}$$
$$\quad init(sz)$$
$$\quad \{ \ast_{k \in [0,sz)} k \mapsto_{\mathsf{Map}} 0 \}$$

$$\forall k\, v. \{k \mapsto_{\mathsf{Map}} v\}$$
$$\quad get(k)$$
$$\quad \{r.\, r = v \wedge k \mapsto_{\mathsf{Map}} v\}$$

$$\forall k\, w\, v. \{k \mapsto_{\mathsf{Map}} w\}$$
$$\quad set(k,v)$$
$$\quad \{k \mapsto_{\mathsf{Map}} v\}$$

$$\forall k\, w. \{k \mapsto_{\mathsf{Map}} w\}$$
$$\quad set\_by\_user(k)$$
$$\quad \{\exists v.\, k \mapsto_{\mathsf{Map}} v\}$$

Figure 3: Pre/Post-conditions of S_{Map}

- `pending` is a unique, unforgeable token, and it gets consumed when calling `init`.
- Now, the following ***conditional contextual refinement*** holds:

$$S_{\mathrm{Map}} \vDash I_{\mathrm{Map}} \sqsubseteq A_{\mathrm{Map}},$$

which means that the refinement

$$I \sqsubseteq A$$

holds under the separation logic conditions $S : \mathtt{String} \rightharpoonup \mathtt{Cond}$.
- `Cond` is the set of pre/post-conditions in separation logic.
- Enjoys benefits of both sides: modular reasoning on shared states, and transitive composition.

Wrapper

In CCR, a conditional refinement is defined as a contextual refinement:

$$S \vDash I \sqsubseteq A \quad \triangleq \quad I \sqsubseteq_{\mathrm{ctx}} \langle S \vdash A \rangle,$$

where $\langle S \vdash A \rangle$ is called a (separation logic) **wrapper**, which converts A into a module that self-enforces the pre/post-conditions of S at the points where A interacts with its program context.

- **Good:**
    - (1) the definition is simple and universal; and
    - (2) we can exploit all the existing benefits of unconditional refinement: horizontal compositionality, vertical compositionality (i.e., transitivity).
- **Question:** Since A and $\langle S \vdash A \rangle$ are modules, the pre/post-conditions in A should be operationalized. Then, what should be the definition of

    OPERATIONALIZED_CONDITIONS?

Stateless Conditional Refinement

```
1  def exp(x: int, n: int) =
2    if n == 0 then
3      return 1
4    else
5      return x * exp(x, n - 1)
```

Figure 4: Module I_{expn}.

```
1  def exp(x: int, n: int) =
2    assume(n >= 0)
3    var r := x ^ n
4    assert(r = x ^ n)
5    return r
```

Figure 5: Module $\langle S \vdash A_{\text{expn}} \rangle$.

$$I_{\text{expn}} \sqsubseteq_{\text{ctx}} \langle S \vdash A_{\text{expn}} \rangle.$$

$$\frac{P \to (\mathbb{T} \lesssim \mathbb{S})}{\mathbb{T} \lesssim (\texttt{assume}(P); \mathbb{S})} \ \text{ASMR} \qquad \frac{P \land (\mathbb{T} \lesssim \mathbb{S})}{\mathbb{T} \lesssim (\texttt{assert}(P); \mathbb{S})} \ \text{ASTR}$$

$$\frac{P \land (\mathbb{T} \lesssim \mathbb{S})}{(\texttt{assume}(P); \mathbb{T}) \lesssim \mathbb{S}} \ \text{ASML} \qquad \frac{P \to (\mathbb{T} \lesssim \mathbb{S})}{(\texttt{assert}(P); \mathbb{T}) \lesssim \mathbb{S}} \ \text{ASTL}$$

Table 2: Encoding conditional wrappers.

Stateless Conditional Refinement

```
1  def exp ( x: int , n: int ) =
2    if n == 0 then
3      return 1
4    else
5      return x * exp ( x, n - 1)
```

Figure 4: Module I_{expn}.

```
1  def exp ( x: int , n: int ) =
2    assume ( n >= 0)
3    var r := x ^ n
4    assert ( r = x ^ n)
5    return r
```

Figure 5: Module $\langle S \vdash A_{\text{expn}} \rangle$.

$$I_{\text{expn}} \sqsubseteq_{\text{ctx}} \langle S \vdash A_{\text{expn}} \rangle.$$

```
1  def main () =
2    var r := exp (3 , 2)
3    // ... r ...
```

Figure 6: Module I_{client}.

```
1  def main () =
2    assert (2 >= 0)
3    var r := exp (3 , 2)
4    assume ( r = 3 ^ 2)
5    // ... r ...
```

Figure 7: Module $\langle S \vdash A_{\text{client}} \rangle$.

$$I_{\text{client}} \sqsubseteq_{\text{ctx}} \langle S \vdash A_{\text{client}} \rangle.$$

Stateful Conditional Refinement

- What is an **ownership**?
  An exclusive control of **resources**, clearly defining who can read or write memory at a given point.
- Why is it important?

$$\because \text{Separation Logic} = \text{Hoare Logic} + \text{Ownership}.$$

- **Question:** How do we transfer resources operationally?
  1. First Attempt: **pass as arguments (and return)**
     However, because $S \vDash I \sqsubseteq A$ is defined as

     $$I \sqsubseteq_{\text{ctx}} \langle S \vdash A \rangle,$$

     the un-conditionality of $\sqsubseteq_{\text{ctx}}$ implies that the resources cannot be passed explicitly between $\langle S \vdash A \rangle$ and another module $\langle S \vdash A' \rangle$.
  2. Solution: **pass resources implicitly**, using the
     dual non-determinism.

Interaction Tree

Let $E : \textbf{Type} \to \textbf{Type}$ be given, where $E(X)$ is called a type of **events** for each $X : \textbf{Type}$. An **interaction tree** i of the type $\texttt{itree}\ E\ T$ can be seen as an open small-step semantics that can:

(i) take a silent deterministic step ($i = \texttt{Tau}\ i'$);

(ii) terminate with a return value, say r, of type T ($i = \texttt{Ret}\ r$); or

(iii) trigger an event, say e, in $E(X)$ for some $X : \textbf{Type}$, and continue execution, say k, for each possible return value in X ($i = \texttt{Vis}\ X\ e\ k$).

Since $\texttt{itree}\ E$ forms a monad for any $E : \textbf{Type} \to \textbf{Type}$, we henceforth use the monad notations $x \leftarrow i; k\ x$ and $i \ggeq k$ for **bind** and $\texttt{ret}$ for **pure**:

- $\texttt{ret}\ r \triangleq \texttt{Ret}\ r.$
- $\texttt{Ret}\ r \ggeq k \triangleq k\ r$
- $\texttt{Tau}\ i \ggeq k \triangleq \texttt{Tau}\ (i \ggeq k).$
- $\texttt{Vis}\ X\ e\ k \ggeq k' \triangleq \texttt{Vis}\ X\ e\ (\lambda x, k\ x \ggeq k')$

Trace

A **trace** is a finite or infinite sequence of `ObsEvent`s (i.e., pairs of an observable event and its return value) that can possibly end one of the four cases:

(i) **Normal termination** (`Term` v).
   A normal termination with a value v of the type `Any`.

(ii) **Silent divergence** (`Diverge`).
   A silent divergence without producing any events.

(iii) **Erroneous termination** (`Error`).
   An erroneous termination is terminating due to an error in the program.

(iv) **Partial termination** (`Partial`).
   A partial termination is due to the user.

Note that:

- `Any` can be understood as the set of all mathematical values;
- the type of traces is denoted by `Trace`; and
- partial termination will serve as a dual of erroneous termination.

Behavior

(1) The behavior of an itree i is defined as $\mathrm{beh}(i)$, where the function $\mathrm{beh} : \mathtt{itree}\ E_\mathrm{p}\ \mathtt{Any} \to \wp(\mathtt{Trace})$ has the following properties:
   - **Prefixed-closed**: $t_0 +\!\!+ t_1 \in \mathrm{beh}(i) \implies t_0 +\!\!+ \mathtt{Partial} \in \mathrm{beh}(i)$.
   - **Postfix-closed**: $t_0 +\!\!+ \mathtt{Error} \in \mathrm{beh}(i) \implies t_0 +\!\!+ t_1 \in \mathrm{beh}(i)$.

(2) The behavior of a function can be defined because its body is an itree.

(3) The behavior of a module can be defined because it consists of initial values of local states and the functions.

(4) The behavior of a closed program can be defined because it consists of several modules.

Remarks.
- $\mathtt{Partial} \in \mathrm{beh}(i)$ holds for any itree i.
- An itree is said to be **divergent** if its execution is a silent divergence without producing any events.
- If an itree i is divergent then $\mathtt{Diverge} \in \mathrm{beh}(i)$.
- $\mathtt{Term}\ r \in \mathrm{beh}(\mathtt{Ret}\ r)$.
- $\mathrm{beh}(i) \subseteq \mathrm{beh}(\mathtt{Tau}\ i)$.

Dual Non-determinism

- What is the "dual non-determinism"?
  It refers to either the ***demonic non-determinism***, or its dual ***angelic non-determinism***.

- What do the terms "demonic non-determinism" and "angelic non-determinism" mean, respectively?
  In CCR, they refer to `choose` and `take`, respectively, where:

Syntax	Semantics	Analogy
$\texttt{choose}(X)$	$\operatorname{beh}(\texttt{choose}(X) \ggg k) \triangleq \bigcup_{x \in X} \operatorname{beh}(k(x))$	$\texttt{assert}(P)$
$\texttt{take}(X)$	$\operatorname{beh}(\texttt{take}(X) \ggg k) \triangleq \bigcap_{x \in X} \operatorname{beh}(k(x))$	$\texttt{assume}(P)$

- Why does it resolve the problem?
  1. The caller "`assert`s" or provides a resource at a call site.
  2. The callee "`assume`s" or receives a resource provided by the caller.
  3. At the point of `return`, the callee `assert`s that it is giving back a resource satisfying its postcondition.
  4. Finally, the caller `assume`s the postconditioin of the callee and receives the resource implicitly.

UB & NB

- What is UB?
  In general, it refers to an "undefined behavior". In CCR,

$$\text{UB} \triangleq \texttt{take}(\emptyset).$$

  In order to prove $\mathbb{T} \precsim \text{UB}$, it is not required for $\mathbb{T}$ to satisfy any particular property—that is, the behavior of the target $\mathbb{T}$ is undefined.

  Therefore, we can accept the definition of UB in CCR.

- What is NB?
  Being different from "unspecified behavior", NB is just the dual of UB:

$$\text{NB} \triangleq \texttt{choose}(\emptyset) = \{\texttt{Partial}\}.$$

  Because it is rare to define the notion of "no behavior", it is also hard to imagine NB. Hence, the only method to figure out NB is to understand the duality of UB and NB—e.g.,

$$\text{NB} \precsim \mathbb{S} \iff \top \iff \mathbb{T} \precsim \text{UB}.$$

WET (Wrapper Elimination Theorem)

- **Question:** How can we remove those operationalized conditions at the end of proving an entire closed program—i.e., the result of inlining all functions of the program in the main function? We need a lemma of the following form:

$$\frac{I_1 \circ I_2 \circ \cdots \circ I_n \sqsubseteq_{\text{beh}} \langle S \vdash A_1 \rangle \circ \langle S \vdash A_2 \rangle \circ \cdots \circ \langle S \vdash A_n \rangle}{I_1 \circ I_2 \circ \cdots \circ I_n \sqsubseteq_{\text{beh}} A_1 \circ A_2 \circ \cdots \circ A_n}$$

- The **_Wrapper Elimination Theorem_** is a foundational soundness theorem for the CCR verification framework, which states as follows:

$$\langle S \vdash A_1 \rangle \circ \langle S \vdash A_2 \rangle \circ \cdots \circ \langle S \vdash A_n \rangle \sqsubseteq_{\text{beh}} A_1 \circ A_2 \circ \cdots \circ A_n.$$

It ensures that wrappers enforcing separation logic conditions can be safely removed after verifying a whole program. This makes conditional refinement practically useful–allowing conditions to be operationalized initially but eliminated safely and entirely at runtime.

Contribution

CCR:

- Is formalized in Coq.
- Includes challenging case studies:
    1. shared memory;
    2. mutual recursion;
    3. function pointers;
    4. termination/non-termination; and
    5. system calls.
- Is ready to be used:
    1. end-to-end verification (with CompCert), and
    2. executable (with Interaction Trees).
- Inspired "DimSum".
- Extended to "Concurrency + CCR".
- Is used by "Program logic for int2ptr casting".
- Motivated "Operationalizing liveness".

Thank you for listening!