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	$\{P\}\,C\,\{Q\} \begin{cases} P: & \text{precondition,} \\ C: & \text{code,} \\ Q: & \text{postcondition.} \end{cases}$	modular reasoning about shared state
Refinement	$I \sqsubseteq S \begin{cases} I: & \text{implementation,} \\ S: & \text{specification.} \end{cases}$	transitive proof composition

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

Summary.

- ***Separation logic*** supports both features.
- ***Refinement*** supports both features.

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

Marrying Separation Logic and Refinement

1. **Relational Separation Logic.**
   - Implemented in ***Simuliris***.
   - 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

```
private data := NULL
def init(sz: int) =
    data := calloc(sz)
def get(k: int): int =
    return *(data + k)
def set(k: int, v: int) =
    *(data + k) := v
def set_by_user(k: int) =
    set(k, input())
```

Listing 1: Module I_{Map}.

```
private map := (fun k => 0)
def init(sz: int) =
    skip
def get(k: int): int =
    return map[k]
def set(k: int, v: int) =
    map := map[k<-v]
def set_by_user(k: int) =
    set(k, input())
```

Listing 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

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

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

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

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

Figure 1: 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:

  $$S_{\text{Map}} \models I_{\text{Map}} \sqsubseteq A_{\text{Map}},$$

  which means that the refinement

  $$I \sqsubseteq A$$

  holds under the separation logic conditions $S : \texttt{String} \rightharpoonup \texttt{Cond}$.
- `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, 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

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

Listing 3: Module I_{expn}.

```
def exp(x: int, n: int) =
    assume(n >= 0)
    var r := x ^ n
    assert(r = x ^ n)
    return r
```

Listing 4: 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

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

Listing 5: Module I_{expn}.

```
def exp(x: int, n: int) =
    assume(n >= 0)
    var r := x ^ n
    assert(r = x ^ n)
    return r
```

Listing 6: Module $\langle S \vdash A_{\text{expn}} \rangle$.

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

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

Listing 7: Module I_{client}.

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

Listing 8: 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?
  (i) 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 $\langle S \vdash A' \rangle$.
  (ii) Solution: **pass resources implicitly**, using the
  dual non-determinism.

Interaction Tree

Let $E : \mathbf{Type} \to \mathbf{Type}$ be given, where $E(X)$ is called a type of ***events*** for each $X : \mathbf{Type}$. 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 r of the type T ($i = \texttt{Ret}\ r$); or

(iii) trigger an event e in $E(X)$ for some $X : \mathbf{Type}$, and resume execution with continuation k for each possible input value of the type X ($i = \texttt{Vis}\ X\ e\ k$).

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

$$\begin{aligned} \texttt{ret}\ r &\triangleq \texttt{Ret}\ r. \\ \texttt{Ret}\ r \gg= k &\triangleq k\ r, \\ \texttt{Tau}\ i \gg= k &\triangleq \texttt{Tau}\ (i \gg= k), \\ \texttt{Vis}\ X\ e\ k \gg= k' &\triangleq \texttt{Vis}\ X\ e\ (\lambda x, k\ x \gg= k'). \end{aligned}$$

Trace

A ***trace*** is a finite or infinite sequence of `ObsEvents` (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).
Normal termination with a value v of the type `Any`.

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

(iii) ***Erroneous termination*** (`Error`).
Termination due to an error in the program.

(iv) ***Partial termination*** (`Partial`).
Partial termination due to the user interrupting execution.

Note that:

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

Behavior

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

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

(3) The behavior of a module can be defined because it consists of initial values of local states and functions(which themselves are defined as itrees).

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

Remarks.

- An itree is said to be ***divergent*** if its execution diverges silently without producing any events. If an itree i is divergent then $\texttt{Diverge} \in \text{beh}(i)$.
- $\texttt{Term}\ r \in \text{beh}(\texttt{Ret}\ r)$; $\texttt{beh}(i) \subseteq \text{beh}(\texttt{Tau}\ i)$; and $\texttt{Partial} \in \text{beh}(i)$.

Dual Non-determinism

- What is “dual non-determinism”?
  It refers to either ***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
$\texttt{choose}(X)$	$\mathrm{beh}(\texttt{choose}(X) \gg= k) \triangleq \bigcup_{x \in X} \mathrm{beh}(k(x))$	$\texttt{assert}(P)$
$\texttt{take}(X)$	$\mathrm{beh}(\texttt{take}(X) \gg= k) \triangleq \bigcap_{x \in X} \mathrm{beh}(k(x))$	$\texttt{assume}(P)$

- Why does it resolve the problem?
  1. The caller’s “`assert`” provides a resource at a call site.
  2. The callee’s “`assume`” receives the 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 postcondition of the callee and receives the resource implicitly.

UB & NB

- What is $\texttt{UB}$?
  In general, it refers to "undefined behavior". In CCR,

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

  In order to prove $\mathbb{T} \lesssim \texttt{UB}$, 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 $\texttt{UB}$ in CCR.

- What is $\texttt{NB}$?
  Being different from "unspecified behavior", $\texttt{NB}$ is just the dual of $\texttt{UB}$:

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

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

$$\texttt{NB} \lesssim \mathbb{S} \iff \top \iff \mathbb{T} \lesssim \texttt{UB}.$$

WET (Wrapper Elimination Theorem)

- **Question:** How can we remove those operationalized conditions after 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!