From Separation Logic to Staged Logic and Beyond

Wei-Ngan Chin
Dept of Computer Science
National University of Singapore

Research Done

1990-1999 : Program Transformation

2000-2006 : Advanced Type Systems

2007-2019 : SLEEK/HIP (Separation Logic)

2020- Temporal Effects/Incorrectness Logic

- Staged Logic for HO-Functions
- Continuations (Alge Effects + Shift/Reset)
- Concurrency Verification (re-looking)
- Type Safety via Hoare Logic (in progress)

Research Highlights

Data structures verification via Separation Logic (VMCAI07)

Pre/Post Specification for Loops (in HIP/SLEEK)

Termination and Non-Termination Specification (ICFEM15,PLDI15)
Immutability Specification (OOPSLA11)
Specification Inference (ASIAN06, SAC10, CAV14, APLAS13)

Concurrency Verification (PEPM15, TASE23)

Staged Logics for Higher-Order Functions (FM24, ICFP24)

Hoare Logic for Type-Safety and Race Freedom (draft paper)
Hoare Logic for Bug Finding (in progress)

Verification via

Separation Logic

VMCAI07

Separation Logic Basics

data cell { int val }

points-to

pre-condition

post-condition

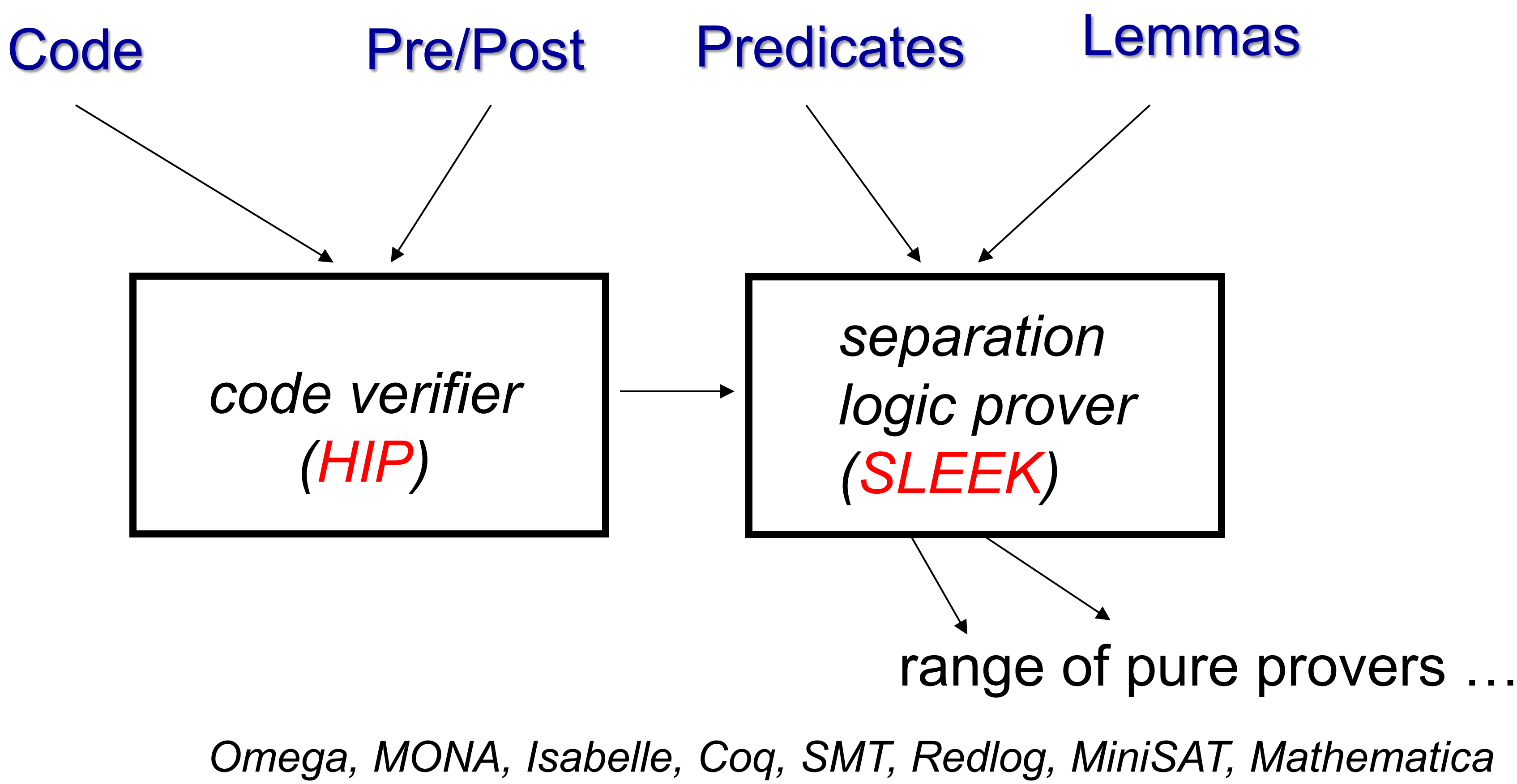
$\{x \mapsto cell(a) * y \mapsto cell(b)\}$ swap(x,y) $\{x \mapsto cell(b) * y \mapsto cell(a)\}$

code

spatial conjunction

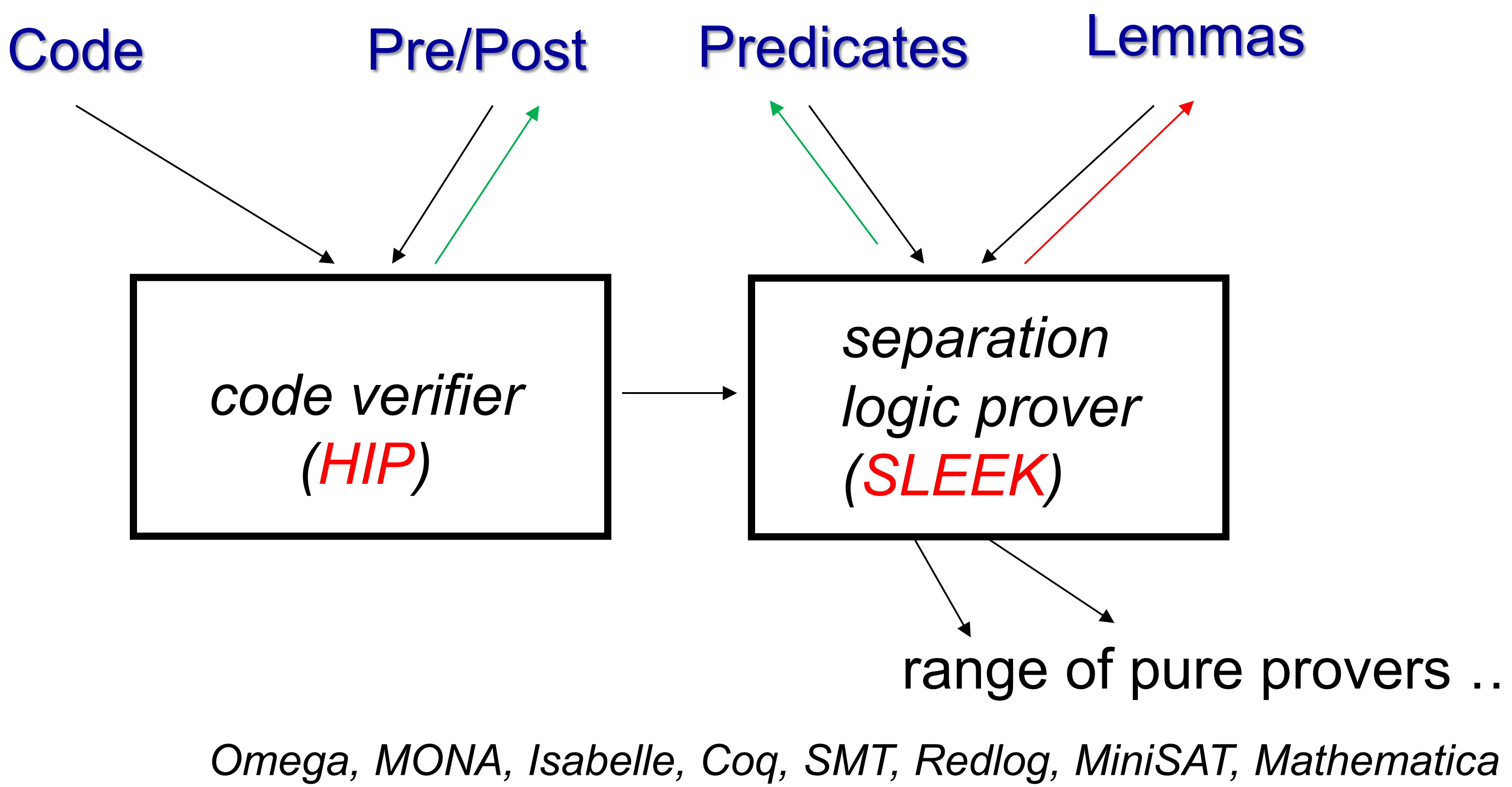
$\{x \mapsto cell(a) \wedge x=y\}$ swap(x,y) $\{x \mapsto cell(a) \wedge x=y\}$

Verification System



Inference System

[CAV14,APLAS13]
[ASIAN06,CAV11]



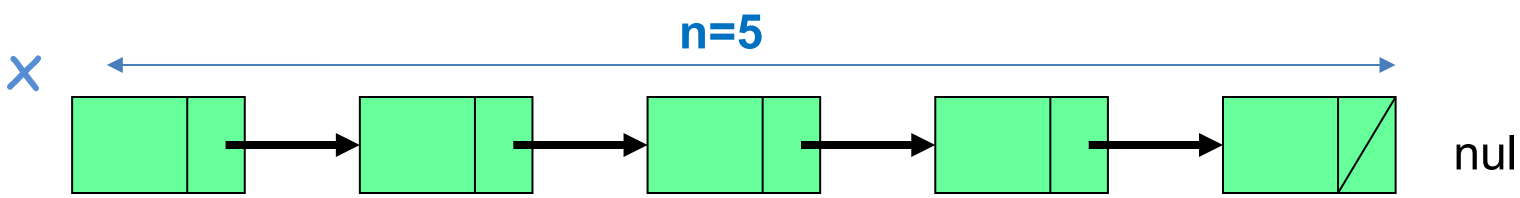
Abstraction via

Inductive Predicates

Predicate : Linked-List with Size

data node { int val; node next }

Example of Singly Linked List : ll(x,n)



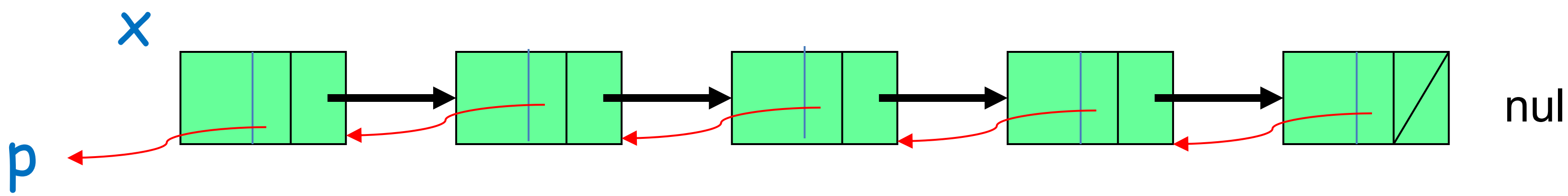
$$ll(x,n) \equiv x{=}null \wedge n{=}0$$
$$\vee \;\; \exists\, q \,.\, x \mapsto node(\_,q) * ll(q,n{-}1)$$
$$inv\ n \geq 0 \;;$$

Flexible abstraction – captures just acyclic shape and length of list here

Doubly Linked-List

data node2 { int val; node2 prev, node2 next }

Example of Doubly Linked List : dll(x,p)



$$\mathrm{dll}(x,p) \equiv x{=}\mathrm{null}$$
$$\vee\ \exists\, q\,.\, x \mapsto \mathrm{node2}(\_,p,q) * \mathrm{dll}(q,x)$$

Handles must-aliasing well ..

AVL Tree

data node2 { int val; node2 prev, node2 next }

$avl(x,h) \equiv x=null \wedge h=0$

$\vee\ \exists\, p,q\ .\ x \mapsto node2(\_,p,q) * avl(p,h_1) * avl(q,h_2)$

$\wedge\ h=1+max(h_1, h_2) \wedge -1 \leq h_1 - h_2 \leq 1$

$inv\ h \geq 0$

Above captures height and near-balancing.

How can sortedness be captured ?

AVL Tree (non-empty)

Sortedness can be captured with
(i) two extra parameters, and
(ii) use of non-empty AVL trees

$$\mathrm{avl}(x,h,mn,mx) \equiv x \mapsto \mathrm{node2}(mn,null,null) \wedge h=1 \wedge mn=mx$$
$$\vee\ \exists\, p,q\,.\ x \mapsto \mathrm{node2}(mx,p,null) * \mathrm{avl}(p,h_1,mn,mx_1) \wedge h=1+h_1 \wedge mx_1 \le mx$$
$$\vee\ \exists\, p,q\,.\ x \mapsto \mathrm{node2}(mn,null,q) * \mathrm{avl}(q,h_2,mn_2,mx) \wedge h=1+h_2 \wedge mn \le mn_2$$
$$\vee\ \exists\, p,q\,.\ x \mapsto \mathrm{node2}(v,p,q) * \mathrm{avl}(p,h_1,mn,mx_1) * \mathrm{avl}(q,h_2,mn_2,mx) \wedge h=1+\max(h_1,h_2) \wedge mx_1 \le v \le mn_2$$
$$\mathbf{inv}\ h \ge 1 \wedge mn \le mx$$

AVL Tree (possibly empty)

Alternatively, we can more succinctly capture it using fictional min and max values
(i) fictional min and max values;
(ii) trees with a possibly null scenario.

$$avl(x,h,mn,mx) \equiv .. \boxed{x{=}null \wedge h{=}0 \wedge mn{=}\infty \wedge mx{=}{-}\infty}$$

$$\vee\ \exists\, p,q\ .\ x \mapsto node2(v,p,q) * avl(p,h_1, mn_1,mx_1)$$

$$* \ avl(q,h_2,mn_2, mx_2) \wedge h{=}1{+}max(h_1,h_2) \wedge mx_1 \leq v \leq mn_2$$

$$\wedge\ mn{=}min(mn_1,v) \wedge mx{=}max(mx_2,v)$$

$$inv\ h{\geq}0 \wedge (h{>}0 \rightarrow min \leq max)$$

Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, Wei-Ngan Chin: **Certified Reasoning with Infinity.** FM 2015: 496-513

Modular Verification

Code - A Function and its Loop

```
int length(node xs)


{ int n=0;
  while (xs!=null)


  { xs = xs.next;
    n = n+1 };
  return n;
}
```

Adding Pre/Post Specs

```
int length(node xs)
// req    ll<xs,m>
// ens[r] ll<xs,m> ∧ r=m;
{ int n=0;
  while (xs!=null)
  // req ll<xs,m>
  // ens ll<xs,m> ∧ xs'=null ∧ n'=n+m;
  { xs = xs.next;
    n = n+1 };
  return n;
}
```

per-method spec

pre/post for loops

xs, **n**= original values at pre

xs', **n'**= latest values

Other Improvements?

- Immutabilty annotation (OOPSLA11)
- Termination/Non-Termination (PLDI15)
- Structured Specification (FM11)

Immutable Borrow Annotation

```
int length(node xs)
// req    ll<xs,m>@I
// ens[r]  r=m;
{ int n=0;
  while (xs!=null)
   // req ll<xs,m>@I
   // ens xs'=null ∧ n'=n+m;
   { xs = xs.next;
     n = n+1 };
  return n;
}
```

@I to indicate immutable (read-only) borrow

Benefit

- Precise and Concise
- Functional Correctness

Cristina David, Wei-Ngan Chin:
Immutable specifications for more concise and precise verification. OOPSLA 2011: 359-374

Termination/Non-Termination

```
int length(node xs)
 // req    ll<xs,m>@I ∧ Term[]
 // ens[r]  r=m;
 // req    clist<xs>@I ∧ Loop
 // ens[r]  false;
{ int n=0;
  while (xs!=null)
    // req ll<xs,m>@I ∧ Term[m]
    // ens xs'=null ∧ n'=n+m;
    // req  clist<xs>@I ∧ Loop
    // ens  false;
  { xs = xs.next; n = n+1 };
  return n;
}
```

Term[..] *for termination proving*
Loop *for non-termination proving*
MayLoop *for unknown status [PLDI15]*

structured spec [FM11]

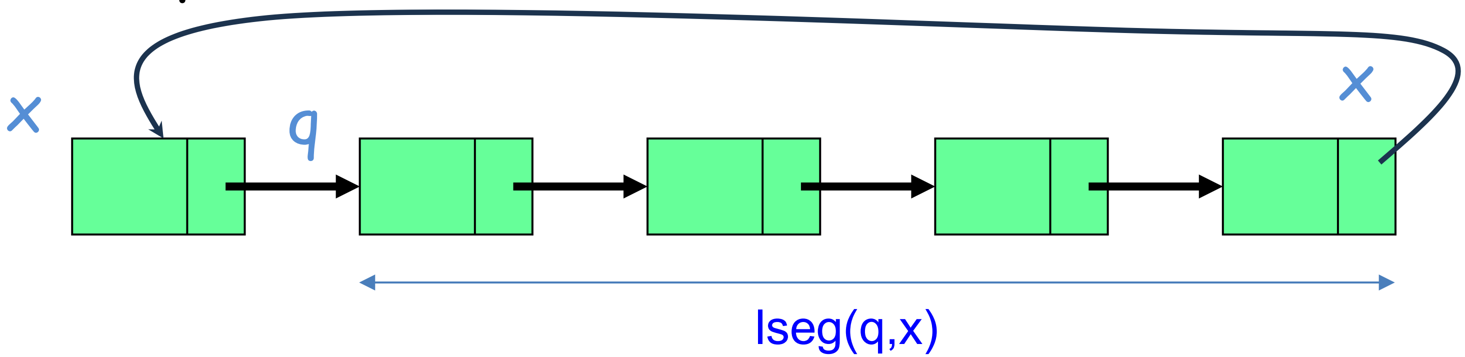
Ton Chanh Le, Shengchao Qin, Wei-Ngan Chin:
Termination and non-termination specification inference. PLDI 2015: 489-498

Cristian Gherghina, Cristina David, Shengchao Qin, Wei-Ngan Chin:
Structured Specifications for Better Verification of Heap-Manipulating Programs. FM 2011: 386-401

Circular List

data node { int val; node next }

Example of Circular List : clist(x)



$clist(x) \equiv x \mapsto node(\_,q) * lseg(q,x)$ inv x!=null ;

$lseg(x,p) \equiv x=p \vee x \mapsto node(\_,q) * lseg(q,p) \wedge x!=p$

lemma $lseg(x,q) * q \mapsto node(\_,x) \Rightarrow clist(q)$

Summary of SLEEK/HIP/Heifer

- High degree of automation
- Proof search though lemmas, multi-specs
- Leverage on existing pure provers
- Support for Inference (CAV14,APLAS13)
- Support for Concurrency Reasoning (PEPM15,..)
- Support for Higher-Order Functions (FM24,..)
- Hoare Logic for Bug Confirmation (on-going)
- Hoare Logic for Type Safety (on-going)

Higher-Order Functions [FM24]

Staged Specification Logic for Verifying Higher-Order Imperative Programs

Darius Foo(✉)[0000−0002−3279−5827] Yahui Song[0000−0002−9760−5895]
Wei-Ngan Chin[0000−0002−9660−5682]

School of Computing, National University of Singapore, Singapore
{dariusf,yahuis,chinwn}@comp.nus.edu.sg

Abstract. Higher-order functions and imperative states are language features supported by many mainstream languages. Their combination is expressive and useful, but complicates specification and reasoning, due to the use of yet-to-be-instantiated function parameters. One inherent limitation of existing specification mechanisms is its reliance on *only two stages* : an initial stage to denote the precondition at the start of the method and a final stage to capture the postcondition. Such two-stage specifications force *abstract properties* to be imposed on unknown function parameters, leading to less precise specifications for higher-order methods. To overcome this limitation, we introduce a novel extension to Hoare logic that supports *multiple stages* for a call-by-value higher-order language with ML-like local references. Multiple stages allow the behavior of unknown function-type parameters to be captured abstractly as uninterpreted relations; and can also model the repetitive behavior of each recursion as a separate stage. In this paper, we define our staged logic with its semantics, prove its soundness and develop a new automated higher-order verifier, called Heifer, for a core ML-like language.

Effectful higher-order programs
Concurrency reasoning potential

What is Separation Logic?

$D ::= \mathrm{emp} \mid x \mapsto t(..) \mid D_1 * D_2 \mid D_1 \vee D_2$
$\mid \mathrm{pred}(..)$
$\mid D \wedge \mathrm{pure}$

Pre/Post Spec

$\mathrm{req}\ D_{pre}\ \ \mathrm{ens}[r]\ D_{post}$

What is Staged Logic?

$S ::= \mathrm{req}\ D \mid \mathrm{ens}[r]\ D \mid S_1 ; S_2 \mid f(v^*) \mid S_1 \vee S_2$
$\mid S_1 \wedge S_2$ // multi pre/post
$\mid S_1 * S_2 \mid \ldots$ // for concurrency

Staged Logic for Methods

update(x,n) = x:=!x+n

update ::: req $x \mapsto a$; ens[r] $x \mapsto a+n \wedge r=()$

cas(x,v,n)

points-to without (cell) type constructor

cas ::: req $x \mapsto a$; ens[r] $x \mapsto n \wedge r \wedge a=v \ \vee\ x \mapsto a \wedge \neg r \wedge a!=v$

How about higher-order functions?

compose(g,f,x) = g(f(x))

compose ::: ens[r] $\exists$ a . f(x,a); g(a,r)

Hoare Rules withStaged Logics

$$\frac{\Phi_1 \sqsubseteq \Phi_3 \quad \{\ \Phi_3\ \}\ e\ \{\ \Phi_4\ \} \quad \Phi_4 \sqsubseteq \Phi_2}{\{\ \Phi_1\ \}\ e\ \{\ \Phi_2\ \}}\ \textbf{Conseq} \qquad \frac{\{\ \Phi_1\ \}\ e\ \{\ \Phi_2\ \}}{\{\ \Phi;\Phi_1\ \}\ e\ \{\ \Phi;\Phi_2\ \}}\ \textbf{Frame}$$

$$\frac{}{\{\ \Phi\ \}\ x\ \{\ \Phi\,;\mathbf{ens}[x]\ emp\ \}}\ \textbf{Var} \qquad \frac{fresh\ r \qquad v ::= c \mid nil \mid x_1{::}x_2}{\{\ \Phi\ \}\ v\ \{\ \Phi\,;\exists\, r{\cdot}\,\mathbf{ens}[r]\ r{=}v\ \}}\ \textbf{Val}$$

$$\frac{fresh\ r}{\{\ \Phi\ \}\ ref\ x\ \{\ \Phi\,;\exists\, r{\cdot}\,\mathbf{ens}[r]\ r \mapsto x\ \}}\ \textbf{Ref}$$

$$\frac{fresh\ a, res}{\{\ \Phi\ \}\ !x\ \{\ \Phi\,;\exists\, a, res\cdot \mathbf{req}\, x \mapsto a\,;\mathbf{ens}[res]\ x \mapsto a \wedge res{=}a\ \}}\ \textbf{Deref}$$

$$\frac{}{\{\ \Phi\ \}\ x_1{:=}x_2\ \{\ \Phi\,;\mathbf{req}\, x_1 \mapsto \_\,;\mathbf{ens}[\_]\ x_1 \mapsto x_2\ \}}\ \textbf{Assign}$$

$$\frac{\{\ \Phi;\mathbf{ens}[\_]\ x\ \}\ e_1\ \{\ \Phi_1\ \} \quad \{\ \Phi;\mathbf{ens}[\_]\ \neg x\ \}\ e_2\ \{\ \Phi_2\ \}}{\{\ \Phi\ \}\ if\ \ x\ \ then\ e_1\ \ else\ e_2\ \{\ \Phi_1 \vee \Phi_2\ \}}\ \textbf{If}$$

$$\frac{fresh\ x \quad \{\ \Phi\ \}\ e_1\ \{\ \exists\, r{\cdot}\,\Phi_1[r]\ \} \quad \{\ [r{:=}x]\Phi_1\ \}\ e_2\ \{\ \Phi_2\ \}}{\{\ \Phi\ \}\ let\ \ x{=}e_1\ \ in\ e_2\ \{\ \exists\, x\cdot\Phi_2\ \}}\ \textbf{Let}$$

$$\frac{fresh\ res \quad \{\ \mathbf{ens}[\_]\ Pure(\Phi)\ \}\ e\ \{\ \exists\, r'{\cdot}\,\Phi_p[r']\ \} \quad ([r'{:=}r]\Phi_p) \sqsubseteq \Phi_s}{\{\ \Phi\ \}\ fun\ (x^*)\,\exists\, r{\cdot}\,\Phi_s[r] \to e\ \{\ \Phi\,;\exists\, res{\cdot}\,\mathbf{ens}[res]\ res{=}\lambda\,(x^*, r) \to \Phi_s\ \}}\ \textbf{Lambda}$$

$$\frac{fresh\ r}{\{\ \Phi\ \}\ f(x^*)\ \{\ \Phi\,;\exists\, r{\cdot}\, f(x^*, r)\ \}}\ \textbf{Call} \qquad \frac{}{\{\ \Phi\ \}\ assert\ D\ \{\ \Phi\,;\mathbf{req}\, D@R\ \}}\ \textbf{Assert}$$

Hoare Proof (1) for foo

foo(x) = let () = x:=!x+1 in x:=!x+2

foo(x) ::: req $x \mapsto a$; ens[r] $x \mapsto a+3$

$$\dfrac{\dfrac{}{\{\text{ens emp}\}\ x{:=}!x+1\ \{\text{ens emp};\text{req } x \mapsto a;\text{ens } x \mapsto a+1\}}\,[\text{Call}]}{\{\text{ens emp}\}\ x{:=}!x+1\ \{\text{req } x \mapsto a;\ \text{ens } x \mapsto a+1\}}\,[\text{Conseq}]$$

$$\dfrac{\dfrac{}{\{\text{req } x \mapsto a;\ \text{ens } x \mapsto a+1\}\ x{:=}!x+2\ \{..;\text{req } x \mapsto c;\text{ens } x \mapsto c+2\}}\,[\text{Call}]}{\{\text{req } x \mapsto a;\ \text{ens } x \mapsto a+1\}\ x{:=}!x+2\ \{\text{req } x \mapsto a;\ \text{ens } x \mapsto a+3\}}\,[\text{Conseq}]$$

$$\dfrac{}{\{\text{ens emp}\}\ \text{let } () = x{:=}!x+1 \text{ in } x{:=}!x+2\ \{\text{req } x \mapsto a;\ x \mapsto a+3\}}\,[\text{Let}]$$

Hoare Proof (2) for foo

foo(x) = let () = x:=!x+1 in x:=!x+2

foo(x) ::: req x↦a ; ens[r] x↦a+3

$$\dfrac{}{\{\text{ens } x{\mapsto}a\}\ x{:=}!x{+}1\ \{\text{ens } x{\mapsto}a;\text{req } x{\mapsto}b;\text{ens } x{\mapsto}b{+}1\}}\ [\text{Call}]$$

$$\dfrac{}{\{\text{ens } x{\mapsto}a\}\ x{:=}!x{+}1\ \{\text{ens } x{\mapsto}a{+}1\}}\ [\text{Conseq}]$$

$$\dfrac{}{\{\text{ens } x{\mapsto}a{+}1\}\ x{:=}!x{+}2\ \{\text{ens } x{\mapsto}a{+}1;\text{req } x{\mapsto}c;\text{ens } x{\mapsto}c{+}2\}}\ [\text{Call}]$$

$$\dfrac{}{\{\text{ens } x{\mapsto}a{+}1\}\ x{:=}!x{+}2\ \{\text{ens } x{\mapsto}a{+}3\}}\ [\text{Conseq}]$$

$$\dfrac{}{\{\text{ens } x{\mapsto}a\}\ \text{let } () = x{:=}!x{+}1 \text{ in } x{:=}!x{+}2\ \{\text{ens } x{\mapsto}a{+}3\}}\ [\text{Let}]$$

Hoare Proof for compose

compose(g,f,x) = let y = f(x) in g(y0

compose(g,f,x) ::: $\exists$ y. f(x,y);g(y,r)}

$$\dfrac{\dfrac{}{\{\text{ens emp}\}\ f(x)\ \{\text{ens emp};\ f(x,r_1)\}}\ [\text{Call}]}{\{\text{ens emp}\}\ f(x)\ \{f(x,r_1)\}}\ [\text{Conseq}]$$

$$\dfrac{}{\{f(x,y)\}\ g(y)\ \{f(x,y);g(y,r)\}}\ [\text{Call}]$$

$$\dfrac{}{\{\text{ens emp}\}\ \text{let } y = f(x) \text{ in } g(y)\ \{\exists\, y.\ f(x,y);g(y,r)\}}\ [\text{Let}]$$

Summarizing Higher-Order Calls

compose(g,f,x) = g(f(x))

compose ::: ens[r] ∃ a . f(x,a); g(a,r)

compose(\() . x:=!x+2, \x . x:=!x+1, x)

::: (req x↦a; ens x↦a+1); (req x↦b; ens x↦b+2)

⊑ req x↦a; ens b=a+1; ens x↦b+2

⊑ req x↦a; ens x↦a+3

compose(\() . !x+3, \x . x:=!x+1, x)

::: (req x↦ a; ens x↦ a+1); (req x↦b@I;ens[r]r=b+3)

⊑ req x↦ a; ens x↦ a+1∧b=a+1; ens[r] r=b+3

⊑ req x↦ a; ens[r] x↦ a+1∧r=a+4

Subsumption vs Refinement

Refinement Calculus [Back, Carrol Morgan et al]

$$\text{spec} \sqsubseteq_r \text{mixed}_1 \sqsubseteq_r \ \ldots \sqsubseteq_r \text{mixed}_n \sqsubseteq_r \text{code}$$

Specification Subsumption

$$\text{code} ::: \text{spec}_1 \sqsubseteq \text{spec}_2 \sqsubseteq \ \ldots \ \sqsubseteq \text{spec}_n \sqsubseteq \text{spec}$$

Specification Subsumption

compose(g,f,x) = g(f(x))
compose ::: ens[r] ∃ a . f(x,a); g(a,r)
compose(\() . y:=!y+2, \x . x:=!x+1, x)

free variable

::: req x↦a; ens x↦a+1; req y↦b; ens y↦b+2;

⊑ req x↦a; req y↦b ; ens x↦a+1; ens y↦b+2;
⊑ req x↦a ∗ y↦b ; ens x↦b+2 ∗ y↦b+2;

Specification ***subsumption*** *reduce* to a more abstract form.

Specification Equivalence

Using classical reasoning with pure formulae.

pure formulae

ens emp

$\Leftrightarrow$ (ens π) $\vee$ (ens $\neg\pi$)

$\Leftrightarrow$ (req π) $\wedge$ (req $\neg\pi$)

Specification ***equivalence*** can be used to *simplify* to an equivalent form without losing precision.

Specification Equivalence

$compose(g,f,x) = g(f(x))$

$compose$ ::: $ens[r]\ \exists\ a\ .\ f(x,a);\ g(a,r)$

$compose(\backslash()\ .\ y{:=}!y+2,\ \backslash x\ .\ x{:=}!x+1,\ x)$

::: $(req\ x\mapsto a;\ ens\ x\mapsto a+1);\ (req\ y\mapsto b;\ ens\ y\mapsto b+2);$

$\Leftrightarrow\ (req\ x\mapsto a;\ ens\ x\mapsto a+1);\ req\ x=y \wedge req\ x\neq y\ ;$

$(req\ y\mapsto b;\ ens\ y\mapsto b+2);$

:

$\Leftrightarrow\ req\ x\mapsto a * y\mapsto b\ ;\ ens\ x\mapsto a+1 * y\mapsto b+2;$

$\wedge\ req\ x\mapsto a \wedge x=y\ ;\ ens\ x\mapsto a+3;$

Recursive HO Methods

```
let rec foldr f a l =
  match l with
  | [] => a
  | h :: t =>
    f h (foldr f a t)
```

Our staged logic solution using precise spec

$$
\begin{aligned}
& foldr(f, a, l, rr) = \\
& \qquad \mathbf{ens}[rr]\, l{=}[] \wedge rr{=}a \\
& \quad \vee\ \exists x, r, l_1 .\ \mathbf{ens}[\_]\, l{=}x{::}l_1; \\
& \qquad\qquad\qquad foldr(f, a, l_1, r); f(x, r, rr)
\end{aligned}
$$

Recursion + Exceptions

Problem : Handling Exceptions

```
let foldr_ex3 l = foldr (fun x r -> if x>=0 then x+r
                                    else raise Exc()) l 0
```

Our Solution via Re-Summarization

$$foldr\_ex3(l, res) \sqsubseteq \mathbf{ens}[res]\, allPos(l) \wedge sum(l, res) \;\vee\; (\mathbf{ens}[\_]\, \neg allPos(l); Exc())$$

$$allPos(l) \quad = \quad (l{=}[]) \vee (\exists\, x, l_1 \,.\, l{=}x{::}l_1 \wedge allPos(l_1) \wedge x{\geq}0)$$

Problem : mutation of list

```
let foldr_ex1 l = foldr (fun x r -> let v = !x
                                    in x := v+1; v+r) l 0
```

Our Solution via Re-Summarization

$$
\begin{aligned}
\mathit{foldr\_ex1}(l, res) \sqsubseteq\ & \exists\, xs\,.\,\mathbf{req}\ \mathit{List}(l, xs)\ ;\ \exists\, ys\,.\\
& \mathbf{ens}[res]\ \mathit{List}(l, ys) \wedge \mathit{mapinc}(xs, ys) \wedge \mathit{sum}(xs, res)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{mapinc}(xs, ys) &= (xs{=}[] \wedge ys{=}[]) \vee (\exists\, x, xs_1, ys_1\,.\,xs{=}x{::}xs_1 \wedge ys{=}(x{+}1){::}ys_1\\
&\qquad\qquad \wedge\ \mathit{mapinc}(xs_1, ys_1))\\
\mathit{List}(l, rs) &= (emp \wedge l{=}[]) \vee (\exists\, x, rs_1, l_1\,.\,x \mapsto r * \mathit{List}(l_1, rs_1) \wedge l{=}x{::}l_1 \wedge rs{=}r{::}rs_1)
\end{aligned}
$$

Problem : stronger assertion

```
let foldr_ex2 l = foldr (fun x r -> assert(x+r>=0);x+r) l 0
```

Our Solution via Re-Summarization

$$foldr\_ex2(l, res) \sqsubseteq \mathbf{req}\ allSPos(l)\ ;\ \mathbf{ens}[res]\ sum(l, res)$$

$$allSPos(l) \quad = \quad (l{=}[]) \vee (\exists\, x, r, l_1 \,.\, l{=}x{::}l_1 \wedge allSPos(l_1) \wedge sum(l, r) \wedge r{\geq}0)$$

Algebraic Effects [ICFP24]

Specification and Verification for Unrestricted Algebraic Effects and Handling

Support for monadic coding and system libraries

YAHUI SONG, School of Computing, National University of Singapore, Singapore
DARIUS FOO, School of Computing, National University of Singapore, Singapore
WEI-NGAN CHIN, Department of Computer Science, National University of Singapore, Singapore

Programming with user-defined effects and effect handlers has many practical use cases involving imperative effects. Additionally, it is natural and powerful to use multi-shot effect handlers for non-deterministic or probabilistic programs that allow backtracking to compute a comprehensive outcome. Existing works for verifying effect handlers are restricted in one of three ways: i) permitting multi-shot continuations under pure setting; ii) allowing heap manipulation for only one-shot continuations; or iii) allowing multi-shot continuations with heap-manipulation but under a restricted frame rule.

This work proposes a novel calculus called *Effectful Specification Logic* (*ESL*) to support unrestricted effect handlers, where zero-/one-/multi-shot continuations can co-exist with imperative effects and higher-order constructs. *ESL* captures behaviors in stages, and provides precise models to support invoked effects, handlers and continuations. To show its feasibility, we prototype an automated verification system for this novel specification logic, prove its soundness, report on useful case studies, and present experimental results. With this proposal, we have provided an extended specification logic that is capable of modeling arbitrary imperative higher-order programs with algebraic effects and continuation-enabled handlers.

CCS Concepts: • **Theory of computation → Logic and verification**; **Program specifications**.

Additional Key Words and Phrases: Multi-shot Continuations, Separation Logic, Automated Verification, Effectful Specification Logic

User-defined Effects and Handlers

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

Example taken from “Effect Handlers in Multicore OCaml” slides by KC Sivaramakrishnan.

User-defined Effects and Handlers

This prints: 0 1 2 3 4

return type of perform

effect declaration

suspends current computation

client program

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

computation

effect handler

delimited continuation

handler

resume suspended computation

Example taken from “Effect Handlers in Multicore OCaml” slides by KC Sivaramakrishnan.

Effects for Concurrency

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit


let fork f = perform (Fork f)
let yield () = perform Yield
```

```
let run main =
  ... (* assume queue of continuations *)
  let run_next () =
    match dequeue () with
    | Some k -> continue k ()
    | None -> ()
  in
  let rec spawn f =
    match f () with
     | () -> run_next () (* value case *)
     | effect Yield k -> enqueue k; run_next ()
     | effect (Fork f) k -> enqueue k; spawn f
   in
   spawn main
```

Also, async/await=
promise, lock, condition
suspense (for concurrency
lib cooperations

Example taken from "Effect Handlers in Multicore OCaml" slides by KC Sivaramakrishnan.

Our Solution: Effectful Specification Logic (ESL)

- Fully modular per-method verification (no global assumption)
- Sequencing, φ_1 ; φ_2

result

- Uninterpreted relations for ***unhandled effects*** and unknown functions, $E(x, r)$

input

- Reducible ***try-catch logic constructs***
- *Normalization: compact* each sequence of pre/post stages, via bi-abduction
- Use ***re-summarization*** (lemma) when handling recursive generated effects

$$\boxed{\begin{array}{lll} (ESL) \quad \varphi & ::= & \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^*;\varphi \mid \\ & & E(x,r) \mid f(x^*,r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi \end{array}}$$

$$D, P, Q ::= \sigma \wedge \pi \qquad\qquad \sigma ::= emp \mid x \mapsto y \mid \sigma * \sigma \mid \ldots$$

Motivating Example

```
effect Label: int


let callee () : int
= let x = ref 0 in
  let ret = perform Label in
  x := !x + 1;
  assert (!x = 1);
  ret
```

$::: \exists x\ .\ ens\ x \mapsto 0; Label(ret); req\ x \mapsto a \wedge a=0; ens[ret]\ x \mapsto a+1;$

$\Leftrightarrow \exists x\ .\ ens\ x \mapsto 0; Label(ret); req\ x \mapsto 0; ens[ret]\ x \mapsto 1;$

Motivating Example (zero shot)

```
let zero_shot () : int
```

$$\Phi_{zero\_shot(ret)} = \exists x \ ; \ (x \mapsto 0 \wedge ret = -1, \mathrm{Norm}(ret))$$

```
= match callee () with
| effect Label k -> -1
```

Fig. 4. A zero-shot handler with Spec.

::: $\exists x \ . \ ens[r] \ x \mapsto 0 \wedge r = -1;$

Motivating Example (one-shot)

```
let one_shot () : int
```

$$\Phi_{one\_shot(r)} = \exists x; (x \mapsto r \wedge r = 1, \mathrm{Norm}(r))$$

```
= match callee () with
| effect Label k ->
                continue k 1
```

Fig. 5. Specifications for a one-shot handler.

::: $\exists x$. ens[r] $x \mapsto 1 \wedge r=1$;

Motivating Example (multi-shot)

```
let multi_shot () : int
```

$\Phi_{multi\_shot(\_)} = \textbf{req}\ (\mathit{false})$

```
= match callee () with
| effect Label k ->
  let _ = continue k 1 in continue k 2
```

Fig. 6. Specifications for a multi-shot handler.

::: req false;   // due to possible assertion failure

Static Try-Catch Reduction Rules

$$\frac{(x \rightarrow \Phi_n) \in \mathcal{H}_\Phi}{\textbf{try}[\delta](\mathcal{N}[r])\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}[r]\ ;\ \Phi_n[r/x]}\ [\mathcal{R}\text{-}Normal]$$

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad E \notin dom(\mathcal{H}_\Phi)}{\textbf{try}[\delta](\mathcal{E}\ ;\ \theta)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{E}\ ;\ \textbf{try}[\delta](\theta)\ \textbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Skip]$$

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ f(x^*, r') \qquad (f(y^*, r) = \Phi_f) \in \mathcal{P}}{\textbf{try}[\delta](\mathcal{E}\ ;\ \theta)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \textbf{try}[\delta](\mathcal{N}\ ;\ \Phi_f[x^*/y^*, r'/r]\ ;\ \theta)\ \textbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Unfold]$$

$$\frac{\mathcal{E}{=}\mathcal{N}; E(x,r) \qquad E{\in}dom(\mathcal{H}_\Phi) \qquad (x' \rightarrow \Phi_n){\in}\mathcal{H}_\Phi \qquad \Phi = \theta[r_1]\ ; \Phi_n[r_1/x']}{\textbf{try}[s](\mathcal{E}\ ;\ \theta)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \textbf{try}[s](\mathcal{E}\ \#\ \Phi)\ \textbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Shallow]$$

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad E \in dom(\mathcal{H}_\Phi) \qquad \textbf{try}[d](\theta)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \Phi}{\textbf{try}[d](\mathcal{E}\ ;\ \theta)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \textbf{try}[d](\mathcal{E}\ \#\ \Phi)\ \textbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Deep]$$

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad (E(y)k \rightarrow \Phi) \in \mathcal{H}_\Phi \qquad \Phi' = \Phi[x/y, (\lambda(r, r_c) \rightarrow \Phi[r_c])/k]}{\textbf{try}[\delta](\mathcal{E}\ \#\ \Phi[r_c])\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}\ ;\ \Phi'}\ [\mathcal{R}\text{-}Eff\text{-}Handle]$$

$$\frac{\begin{array}{c}\mathcal{E} = \mathcal{N}\ ;\ f(x^*, r') \qquad (rec\ f(y^*, r) = \Phi_f) \in \mathcal{P} \qquad fst(\Phi_f) \in dom(\mathcal{H}_\Phi) \\ (\textbf{try}[\delta](f(y^*, r)\ \#\ \Phi)\ \textbf{catch}\ \mathcal{H}_\Phi \sqsubseteq \Phi_{inv}) \in \mathcal{P}\end{array}}{\textbf{try}[\delta](\mathcal{E}\ \#\ \Phi_c)\ \textbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}\ ;\ \Phi_{inv}[x^*/y^*, r'/r, \Phi_c/\Phi]}\ [\mathcal{R}\text{-}Lemma\text{-}App]$$

Fig. 22. Reduction Rules for Try-Catch Constructs

Conclusion : Follow-Up Goals

- Put past research results into practice.
- Build practical automated program verifier.
- Contact me if you wish to collaborate with us on the above project ☺

Research Tools Built

HIP/SLEEK: A formal verification system for imperative programs (C/Java/C++)
https://github.com/hipsleek/hipsleek

- Heifer: A formal verification system for higher-order programs (OCaml)
https://github.com/hipsleek/Heifer

- Songbird: An automated theorem prover for Separation Logic
https://songbird-prover.github.io/

Mostly research prototypes.

Formal Semantics for Separation Logic

$$
\begin{array}{lll}
S,h \models \sigma \wedge \pi & \textit{iff} & [\![\pi]\!]_S \text{ and } S,h \models \sigma \\
S,h \models \textit{emp} & \textit{iff} & \textit{dom}(h) = \{\} \\
S,h \models x \mapsto y & \textit{iff} & \textit{dom}(h) = \{S(x)\} \text{ and } h(S(x)) = [\![y]\!]_S \\
S,h \models \sigma_1 * \sigma_2 & \textit{iff} & \exists h_1\, h_2 .\ h_1 \circ h_2 = h \text{ such that } S,h_1 \models \sigma_1 \text{ and } S,h_2 \models \sigma_2
\end{array}
$$

Formal Semantics for Staged Logic

$$
\begin{array}{ll}
S,h \rightsquigarrow S,h_2,Norm(\_) \models \mathbf{req}\,\sigma{\wedge}\pi & \textit{iff}\;\; h{=}h_1{\circ}h_2 \text{ and } S,h_1 \models \sigma{\wedge}\pi \\
S,h \rightsquigarrow S,h,Err \models \mathbf{req}\,\sigma{\wedge}\pi & \textit{iff}\;\; \forall h_1\,.\,h_1{\subseteq}h \Rightarrow S,h_1 \not\models \sigma{\wedge}\pi \\
S,h \rightsquigarrow S,h,R \models \mathbf{req}\,(\sigma{\wedge}\pi)@R & \textit{iff}\;\; S,h \rightsquigarrow S,h_1,R \models \mathbf{req}\,(\sigma{\wedge}\pi) \\
S,h \rightsquigarrow S,h{\circ}h_1,Norm(r) \models \mathbf{ens}[r]\,\sigma{\wedge}\pi & \textit{iff}\;\; S,h_1 \models \sigma{\wedge}\pi \text{ and } dom(h_1){\cap}dom(h){=}\{\} \\
S,h \rightsquigarrow S_1,h_1,R \models f\,(x^*,r) & \textit{iff}\;\; S(f) = \textit{fun}\,(y^*)\,\Phi[r'] \rightarrow e, \\
 & \qquad S,h \rightsquigarrow S_1,h_1,R \models [r'{:=}r][y^*{:=}x^*]\Phi \\
S,h \rightsquigarrow S_1,h_1,R \models \exists\,x\,.\,\Phi & \textit{iff}\;\; \exists v\,.\,S[x{:=}v],h \rightsquigarrow S_1,h_1,R \models \Phi \\
S,h \rightsquigarrow S_2,h_2,R \models \Phi_1;\Phi_2 & \textit{iff}\;\; S,h \rightsquigarrow S_1,h_1,Norm(r) \models \Phi_1, \\
 & \qquad S_1,h_1 \rightsquigarrow S_2,h_2,R \models \Phi_2 \\
S,h \rightsquigarrow S_1,h_1,\top \models \Phi_1;\Phi_2 & \textit{iff}\;\; S,h \rightsquigarrow S_1,h_1,\top \models \Phi_1 \\
S,h \rightsquigarrow S_3,h_3,Norm(r_3) \models \Phi_1 \vee \Phi_2 & \textit{iff}\;\; \exists h_1,h_2,r_1,r_2\,.\,S,h \rightsquigarrow S_1,h_1,Norm(r_1) \models \Phi_1 \\
 & \qquad \text{and } S,h \rightsquigarrow S_2,h_2,Norm(r_2) \models \Phi_2, \text{ and} \\
 & \qquad (S_3,h_3,r_3){\in}\{(S_1,h_1,r_1),(S_2,h_2,r_2)\} \\
S,h \rightsquigarrow S_1,h_1,\top \models \Phi_1 \vee \Phi_2 & \textit{iff}\;\; S,h \rightsquigarrow S_1,h_1,\top \models \Phi_1 \textit{ or } S,h \rightsquigarrow S_1,h_1,\top \models \Phi_2
\end{array}
$$