Modelling and Reasoning about State

Nick Benton
Microsoft Research, Cambridge
Introduction

- Most programming languages are imperative
  - As time progresses, execution steps read and destructively update the state
- This reflects the model of the underlying hardware
  - To which even declarative languages are compiled, so state matters if we care about compiler correctness for them too
- Down at the bottom we just have a (finite, really) state machine, whose behaviour is not terribly hard to specify
  - Our languages, models and logics are abstractions over that machine
    - So we don’t have to deal with the messy details all the time
    - So we can vary the details of the messy details
      - i.e. so we can say things that are independent of the details of the messy details
State is Scary

- Want to be able to reason compositionally at high level, at low level, and relate the two
- Whenever state is involved, compositional reasoning gets tricky
  - State is implicit. In most languages any computation may read and write store without advertising that fact in its interface/type
    - \( f \ 3 = f \ 3 \) does not always evaluate to true
  - Correctness usually depends on some parts of the state not being modified (or only modified in certain ways) in some parts of the program, but talking about “parts of the state” or delimiting “certain ways” in logics/types/models is tricky
    - Aliasing: \( \{ [x]=3 \ \land \ [y]=4 \} \ [y] := 5 \{ [x]=3 \ \land [y]=5 \} \) ???
    - Separation, effects, regions, ownership,…
  - Single-threading of state means always paying attention to ordering of computations and irreversibility of changes
State is Scary (2)

- References are “generative” so we need to reason about freshness and encapsulation (related to above)
  - $\nu n. \nu n'. \lambda f: v \rightarrow o. f n = f n'$
  - $\lambda f: v \rightarrow o. \text{true}$

- Mutable state increases the range of possible behaviours of programs
  - Storing functions allows recursion to be encoded and introduces recursive domain equations in denotational semantics

- Fragile: exactly which operations are allowed affects properties of language in subtle ways
  - Above eqn doesn’t hold if names can be stored

- State is a frequent source of bugs, warts, kludges and security holes in languages and programs
  - Polymorphic generalization in ML
  - Initialization complexity and pervasive nulls
  - Covariant collections
  - Readonly fields containing read/writable collections
  - General terror: Can this be shared? Who’s responsible for this memory? Might this still be null at this point?
  - Hard to parallelize or optimize and inhibits use of higher abstractions (e.g. LINQ)
We’ve been trying to get a grip on state for at least 50 years

- Program logics: Floyd and Hoare through to separation logic and beyond,…
- Denotational models: from Burstall (state as a function from l-values to r-values) to parametric logical relations, indexed monads over functor categories, coalgebras, game semantics,…
- Fancy types and analyses: from Kildall (old-school dataflow) to regions, capabilities, effect systems, shape analysis, ownership, information flow analysis,…
These lectures

- Relational reasoning about while programs
- Semantics of effect systems
- Semantics of a higher-order language with dynamically allocated local state
- Specifying and verifying a low-level allocator
- Specifying and verifying type soundness for a simple compiler
These lectures

Key ideas

- Separation
- Independence
- Encapsulation
- Binary relations instead of unary predicates
- Invariants: what stays the same instead of what changes
- Extensional rather than intensional reasoning
Analysis and Transformations
Aims:

- Want to prove an analysis only infers true properties of programs
  - Factor into
    - soundness of declarative specification of analysis (e.g. as type system or constraint system), and
    - soundness of inference algorithm wrt specification (I’ll ignore this aspect entirely)

- Given the results of the analysis, want to prove that original and transformed program are observationally equivalent
  - Factor into
    - soundness of declarative specification of which transformations are valid, given analysis
    - correctness of a transformation algorithm, which possibly uses extra heuristic information (Ignored here)
What do analysis properties mean?

- Want to show $\vdash P: \phi$ implies $\vdash P: \phi$

- For simple properties, the meaning of $\phi$ will be some kind of set
  - Terms: $\vdash P: \phi$ iff $P \in [\phi]$
  - Denotations: $[P] \in D$, $[\phi] \subseteq D$ and then $\vdash P: \phi$ iff $[P] \in [\phi]$

- But how to define $[\phi]$?

- If an analysis is computable, its behaviour won’t be closed under observational equivalence
  - $\vdash P: \phi$ and $P \sim P'$ but $\not\vdash P': \phi$

- But range of “degrees of extensionality” for $[\phi]$
Compare: Syntactic approach to type soundness

- Show typeability behaves well wrt small-step transitions semantics
- Pro: It’s usually simple
- Con: Everything else:
  - Doesn’t capture what types \textit{mean} – purely syntactic
  - It’s a cheat – you have to modify the operational semantics you first thought of to make things go wrong (get stuck) when policy is violated
  - Ties soundness to the inference system
  - Requires typing rules to be extended to all entities in the operational semantics
  - Not so good for (in)dependency or transformations
  - Doesn’t tell you what the proof obligations are for code written in another language or that is trusted and unchecked
  - Everything done from scratch every time
Intensionality and instrumentation in defining $[\phi]$

- Analyses often described in a very intensional way
  - Does this function always evaluate its argument?
  - Has this variable been assigned to on any path from that program point to this?
- Such properties not modelled in standard semantics
- Define *instrumented* semantics tracking extra information
  - Labelled reductions
  - Traces of reads and writes
- Pro: It’s usually fairly simple
- Con: Everything else
Transformational semantics of properties

- Wand: `This work suggests that the proposition associated with a program analysis can simply be that “the optimization works” ’
- Possibly rather syntactic, especially at coarse grain
- Underinvestigated
- Work of Führmann and of Plotkin & Power suggests a possible algebraic theory of effects and effect-based transformations…
Extensional semantics of properties

- If $P$ and $f(P)$ are equivalent then this follows in a standard semantics.
- And the reason, $[[\phi]]$, why they are equivalent should be too.
- Intensional approach confuses particular analysis systems and the semantics of the information they produce.
- True preconditions for transformations can be expressed perfectly well in standard semantics (“this command does not change the value of $X+Y$”) even if analysis only detects a stronger intensional property (“this command contains no assignments to either $X$ or $Y$”).
- We’ll try to make $[[\phi]]$ closed under contextual equivalence.
- This helps proofs, but also leads to more powerful and modular analyses.
Intensional vs. extensional reasoning

Why is the following valid?

\[
\begin{align*}
X & := 7; \\
Y & := Y + 1; \\
Z & := X;
\end{align*}
\]

Intensional answer

- The only definition of X which reaches the use of X on line 3 is the one on line 1, and the right hand side of that definition does not contain any variable which is assigned along the path consisting of lines 1 and 2

Extensional answer

- Whenever X is evaluated on the last line, its value is 7
Simplified view

Intensional:
\[ \vdash P: \phi \quad \models P: \phi \quad P \sim f(P) \]

Transformational:
\[ \vdash P: \phi \quad \models P: \phi \quad P \sim f(P) \]

Extensional:
\[ \vdash P: \phi \quad \models P: \phi \quad P \sim f(P) \]
\[ \vdash P: \phi \quad \models P: \phi \quad P \sim f'(P) \]
Proving soundness of analysis-based transformations

- Hundreds of papers on analysis algorithms
- Dozens proving correctness of analyses
- A handful proving correctness of transformations

What’s the problem?

- It turns out to be amazingly difficult even to specify interesting transformations
- Intensionality & “stickiness” interact badly with transformation
- Have to take context seriously
Our approach: contextual reasoning

- Interpret analysis properties as (special kinds of) binary relation, not as predicates
- Present analysis and transformation as rules for deriving typed equations in context
  \[ \Gamma \vdash M = M' : A \]
- Completely standard approach in type theory, categorical logic etc. but rare in static analysis
While programs

- Standard syntax and denotational

\[ X \in \mathbb{V} = \{x, y, \ldots\} \]

\[
\text{int exp } \ni E ::= n \mid X \mid E \text{ iop } E
\]

\[
\text{bool exp } \ni B ::= b \mid E \text{ bop } E \mid \text{not } B \mid B \text{ lop } B
\]

\[
\text{com } \ni C ::= \text{skip} \mid X := E \mid C; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C
\]

\[
S = \mathbb{V} \rightarrow \mathbb{Z}
\]

\[
\llbracket E \rrbracket \in S \rightarrow \mathbb{Z}
\]

\[
\llbracket B \rrbracket \in S \rightarrow \mathbb{B}
\]

\[
\llbracket C \rrbracket \in S \rightarrow S_\perp
\]
Dependency, Dead Code and Constants (DDCC)

- **Base types** $\phi_\tau := \{c\}_\tau \mid \Delta_\tau \mid T_\tau$
  - $\llbracket \{c\}_\tau \rrbracket = \{(c, c)\}$
  - $\llbracket \Delta_\tau \rrbracket = \{(x, x) \mid x \in \llbracket \tau \rrbracket\}$
  - $\llbracket T_\tau \rrbracket = \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket$

- **State types** $\Phi := - \mid \Phi, X: \phi_{\text{int}}$
  - $\llbracket - \rrbracket = S \times S$
  - $\llbracket \Phi, X: \phi_{\text{int}} \rrbracket = \llbracket \Phi \rrbracket \cap \{(S, S') \mid (S(X), S'(X)) \in \llbracket \phi_{\text{int}} \rrbracket\}$

- **Entailment** $\leq$ axiomatises inclusion $\subseteq$ on base and state types (depth+width subtyping)
DDCC Judgements

- Expressions $\vdash E \sim E' : \Phi \Rightarrow \phi_{\text{int}}$
  - $\{(f,f')| \forall (S,S') \in [\Phi]. (f \ S, f' \ S') \in [\phi_{\text{int}}]\}$

- Commands $\vdash C \sim C' : \Phi \Rightarrow \Phi'$
  - $\{(f,f')| \forall (S,S') \in [\Phi]. (f \ S, f' \ S') \in [\Phi']_{\perp}\}$

- Core rules:
  - Subtyping, symmetry, transitivity
  - Congruence rules for expressions including relational abstract interpretations of primitive operations
  - Congruence rules for commands…
DDCC core rules for commands

\[ \vdash \text{skip} \sim \text{skip} : \Phi \Rightarrow \Phi \]

\[ \vdash C_1 \sim C'_1 : \Phi \Rightarrow \Phi' \quad \vdash C_2 \sim C'_2 : \Phi' \Rightarrow \Phi'' \]

\[ \vdash (C_1; C_2) \sim (C'_1; C'_2) : \Phi \Rightarrow \Phi'' \]

\[ \vdash E \sim E' : \Phi, X : \phi_{\text{int}} \Rightarrow \phi'_{\text{int}} \]

\[ \vdash X := E \sim X := E' : \Phi, X : \phi_{\text{int}} \Rightarrow \Phi, X : \phi'_{\text{int}} \]

\[ \vdash B \sim B' : \Phi \Rightarrow \Delta_{\text{bool}} \quad \vdash C \sim C' : \Phi \Rightarrow \Phi \]

\[ \vdash (\text{while } B \text{ do } C) \sim (\text{while } B' \text{ do } C') : \Phi \Rightarrow \Phi \]

\[ \vdash B \sim B' : \Phi \Rightarrow \Delta_{\text{bool}} \quad \vdash C_1 \sim C'_1 : \Phi \Rightarrow \Phi' \quad \vdash C_2 \sim C'_2 : \Phi \Rightarrow \Phi' \]

\[ \vdash (\text{if } B \text{ then } C_1 \text{ else } C_2) \sim (\text{if } B' \text{ then } C'_1 \text{ else } C'_2) : \Phi \Rightarrow \Phi' \]
DDCC equations

- Unit & associativity for sequential composition, commuting conversion for conditionals, loop unrolling…

\[ \vdash C_1 \sim C : \Phi \implies \Phi' \quad \vdash C_2 \sim C : \Phi \implies \Phi' \]

- Equivalent branch

\[ \vdash \text{if } B \text{ then } C_1 \text{ else } C_2 \sim C : \Phi \implies \Phi' \]

- Dead assign

\[ \vdash (X := E) \sim \text{skip} : \Phi, X : \phi_{\text{int}} \implies \Phi, X : \mathbb{T}_{\text{int}} \]

- Constancy, known branches, dead while, divergence
Examples

- Constant folding, known branches, dead code:

```plaintext
if X=3 then X := 7
else skip;
Z := X+1
```

```
\sim
\begin{array}{l}
Z := 8 \\
\end{array}
:\Phi, X:\{3\}, Z:T \Rightarrow \Phi, X:T, Z:\{8\}
```

- Slicing:

```plaintext
I := 1;
S := 0;
P := 1;
while I< N do ( 
  S := S+I;
P := P*I;
I := I+1)
```

```
\sim
\begin{array}{l}
I := 1;
P := 1;
while I< N do ( 
P := P*I;
I := I+1)
\end{array}
:\begin{array}{l}
I:T, S:T, P:T, N:\Delta \Rightarrow 
I:\Delta, S:T, P:\Delta, N:\Delta
\end{array}
```
Smith/Volpano Security Types

\[ \gamma, X : \sigma_{\text{int}} \vdash X : \sigma_{\text{int}} \quad \gamma \vdash n : \sigma_{\text{int}} \quad \gamma \vdash b : \sigma_{\text{bool}} \]

\[ \frac{\gamma \vdash E : \sigma_{\text{int}} \quad \gamma \vdash E' : \sigma_{\text{int}}}{\gamma \vdash E \ \text{iop} \ E' : \sigma_{\text{int}}} \]
+ similar for \text{bop} \ and \ \text{lop}

\[ \gamma, X : \sigma_{\text{int}} \vdash E : \sigma_{\text{int}} \]

\[ \frac{\gamma \vdash X : \sigma_{\text{int}} \quad \gamma \vdash X : \text{=} E : \sigma_{\text{com}}}{\gamma, X : \sigma_{\text{int}} \vdash X : \text{=} E : \sigma_{\text{com}}} \]

\[ \frac{\gamma \vdash C : \sigma_{\text{com}} \quad \gamma \vdash C' : \sigma_{\text{com}}}{\gamma \vdash C ; C' : \sigma_{\text{com}}} \]

\[ \gamma \vdash B : \sigma_{\text{bool}} \quad \gamma \vdash C : \sigma_{\text{com}} \quad \gamma \vdash C' : \sigma_{\text{com}} \]

\[ \frac{\gamma \vdash \text{if} \ B \ \text{then} \ C \ \text{else} \ C' : \sigma_{\text{com}}}{\gamma \vdash \text{if} \ B \ \text{then} \ C \ \text{else} \ C' : \sigma_{\text{com}}} \]

\[ \frac{\gamma \vdash B : L_{\text{bool}} \quad \gamma \vdash C : L_{\text{com}}}{\gamma \vdash \text{while} \ B \ \text{do} \ C : L_{\text{com}}} \]

\[ \frac{\gamma \vdash F : L_T \quad \gamma \vdash C : H_{\text{com}}}{\gamma \vdash C : L_{\text{com}}} \]

\[ \frac{\gamma \vdash F : H_T}{\gamma \vdash C : L_{\text{com}}} \]
Smith/Volpano Security Types

- Translation into DDCC:
  - $L_{\tau}^* = \Delta_{\tau}$, $H_{\tau}^* = T_{\tau}$
  - $(\gamma \vdash E : \sigma)^* = \vdash E \sim E : \gamma^* \Rightarrow \sigma^*$
  - $(\gamma \vdash C : L_{\text{com}})^* = \vdash C \sim C : \gamma^* \Rightarrow \gamma^*$
  - $(\gamma \vdash C : H_{\text{com}})^* = \vdash C \sim \text{skip} : \gamma^* \Rightarrow \gamma^*$

- Theorem: If J derivable in S/V then J* derivable in DDCC

- C satisfies strong sequential noninterference (G/M) in context $\gamma$ if $\vdash C \sim C : \gamma^* \Rightarrow \gamma^*$

- This is the semantic property S/V really wanted

- DDCC marginally stronger
Relational Hoare Logic (RHL)

- Define state relations by boolean expressions over variables $X^{(1)}$ and $X^{(2)}$ specifying which state they come from
- Axiomatize when a pair of commands map a prerelation into a postrelation
- Not restricted to partial equivalence relations
- Parameterized by system for deciding entailment and PERness
Core RHL rules

\[
\vdash \text{skip} \sim \text{skip} : \Phi \Rightarrow \Phi
\]

\[
\vdash C \sim C' : \Phi \land (B(1) \land B'(2)) \Rightarrow \Phi' \quad \vdash D \sim D' : \Phi \land \neg(B(1) \lor B'(2)) \Rightarrow \Phi'
\]

\[
\vdash \text{if } B \text{ then } C \text{ else } D \sim \text{if } B' \text{ then } C' \text{ else } D' : \Phi \land (B(1) = B'(2)) \Rightarrow \Phi'
\]

\[
\vdash C \sim C' : \Phi \Rightarrow \Phi' \quad \vdash D \sim D' : \Phi' \Rightarrow \Phi''
\]

\[
\vdash C \ ; \ D \sim C' \ ; \ D' : \Phi \Rightarrow \Phi''
\]

\[
\vdash X := E \sim Y := E' : \Phi[E(1)/X(1), E'(2)/Y(2)] \Rightarrow \Phi
\]

\[
\vdash C \sim C' : \Phi \land (B(1) \land B'(2)) \Rightarrow \Phi \land (B(1) = B'(2))
\]

\[
\vdash \text{while } B \text{ do } C \sim \text{while } B' \text{ do } C' : \Phi \land (B(1) = B'(2)) \Rightarrow \Phi \land \neg(B(1) \lor B'(2))
\]

Plus prerelation strengthening, postrelation weakening and symmetry and transitivity for PERs
RHL equations

- Basic equivalences as before
- Common Branch:
  \[ \vdash C_1 \sim C : \Phi \land B\langle 1 \rangle \Rightarrow \Phi' \quad \vdash C_2 \sim C : \Phi \land \neg B\langle 1 \rangle \Rightarrow \Phi' \]
  \[ \vdash \text{if } B \text{ then } C_1 \text{ else } C_2 \sim C : \Phi \Rightarrow \Phi' \]
- Dead Assign:
  \[ \vdash X := E \sim \text{skip} : \Phi[E\langle 1 \rangle/X\langle 1 \rangle] \Rightarrow \Phi \]
- + Converse versions
- + Dead while, known branch
Examples

- Invariant hoisting:

```plaintext
while I<N do
  X := Y+1;
  I := I+X;
```

~

```plaintext
X := Y+1;
while I<N do
  I := I+X;
```

\[ I(1)=I(2) \land N(1)=N(2) \land Y(1)=Y(2) \]
\[ \Rightarrow \]
\[ I(1)=I(2) \land N(1)=N(2) \land Y(1)=Y(2) \]

- Dead code:

```plaintext
if X>3 then
  Y := X;
else
  Y := 7;
```

~

```plaintext
skip
```

\[ X(1)=X(2) \land Y(1)>2 \land Y(2)>2 \]
\[ \Rightarrow \]
\[ Y(1)>2 \land Y(2)>2 \]
Translating other logics/type systems into RHL

- E.g. DDCC embeddable within RHL. DDCC types are conjunctions of assertions of the form
  \[ X(1) = X(2) \] and \[ X(1) = n \land X(2) = n \]
- Need to index by finite sets of variables to translate ordinary Hoare logic
- Can soundly add squaring of valid total correctness judgements to RHL
- Paper: translation of very naïve system for available expression analysis and removal of redundant evaluation
Related work

- Functional programs: Wand’s group; Amtoft; Damiani & Giannini; Kobayashi; Benton & Kennedy
- Temporal logic: Lacey, Jones, Van Wyk & Frederikson; Lerner, Millstein & Chambers
- Relations: Hunt & Sands, Sabelfeld & Sands, Bannerjee & Naumann; Abadi, Bannerjee, Heintze & Riecke
- Credible compilation: Rinard & Marinov
- Translation validation: Zuck, Pnueli, Fang & Goldberg; Necula
- Kleene algebra: Kozen & Patron
- Relations and Hoare logic: Yang
- Logic of relations: Abadi, Cardelli & Curien
Summary

- Proving the correctness of such simple analysis-based transformations *should* be simple. It is.
- No rocket science
- To relate to classical stuff, need to reformulate on unstructured flowgraphs
- Need to say something about actual analysis and transformation algorithms
  - Sensible place to start: Lacey, Jones, Van Wyk, Frederikson
Extensional Semantics of a Simple Effect System

Joint work with Andrew Kennedy, Martin Hofmann & Lennart Beringer
Reading and writing

- When is this equivalence valid?

\[
C \ ; \ \text{if } B \ \text{then } C_1 \ \text{else } C_2 \\
\cong \\
\text{if } B \ \text{then } \{ C; C_1 \} \ \text{else } \{ C; C_2 \}
\]

Answer: when B’s reads are disjoint from C’s writes.

- We’re interested in effect systems that can validate equations such as this. Specifically, we want to study their semantics.
Effect systems

- Effect systems (Gifford & Lucassen) associate to every term an approximation of its effect, such as:
  - Reading from a location in the store
  - Writing to a location in the store
  - Throwing an exception
  - Performing I/O
  - Allocating an object or ref cell
  - Diverging

- Effect information can be used to justify transformations e.g. in optimizing compilers.
Extensional interpretation of effects

What does it actually *mean* to “(not) read” or “(not) write”? Start simple: a store consisting of just 2 integer locations. Let $f$ be the denotation of a command i.e. $f \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z}$.

Suppose C does not *write* to the first location. Extensionally: there is some $g : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ such that $f(x,y) = (x,g(x,y))$

Suppose C does not *read or write* the first location. Extensionally: there is some $g : \mathbb{Z} \rightarrow \mathbb{Z}$ such that $f(x,y) = (x,g(y))$

Suppose C does not *read* from the first location. Extensionally: there is some $h : \mathbb{Z} \rightarrow \mathbb{B}$, $g_1, g_2 : \mathbb{Z} \rightarrow \mathbb{Z}$ such that $f(x,y) = (h(y) \ ? \ x : g_1(y), g_2(y))$
The “does not read” effect, in particular, was a little intricate to capture. But move to a \textit{relational} interpretation, and we have a much slicker characterization of reading and writing.

\( \Delta = \text{diagonal relation} \), \( \times \) and \( \rightarrow \) usual constructions on relations

\( f : R \) shorthand for \((f,f) \in R\)

\( C \) does not write to the first location:

\( \forall R \subseteq \Delta. f : R \times \Delta \rightarrow R \times \Delta. \)

\( C \) does not read or write the first location:

\( \forall R. f : R \times \Delta \rightarrow R \times \Delta. \)

\( C \) does not read from the first location:

\( \forall R \supseteq \Delta. f : R \times \Delta \rightarrow R \times \Delta. \)
Framework

Base type system
\[ \Gamma \vdash M : A \]

Base semantics
(sets and functions)
\[ \llbracket \Gamma \vdash M : A \rrbracket \]

“erases to”

Refined type system
(effect annotation, subtyping)
\[ \Theta \vdash M : X \]

Refined semantics
(partial equivalence relations over base semantics)
\[ \llbracket \Theta \vdash M : X \rrbracket \]

“in diagonal of”
Base language

- **Types**
  \[ A, B \ := \ \text{unit} \mid \text{int} \mid \text{bool} \mid A \times B \mid A \rightarrow TB \]
  \[ \Gamma \ := \ x_1 : A_1, \ldots, x_n : A_n \]

- **Terms**
  \[ V, W \ := \ () \mid n \mid b \mid (V, W) \mid \lambda X : A. M \mid V + W \mid \pi_i V \mid \ldots \]
  \[ M, N \ := \ \text{val } V \mid \text{let } x \leftarrow M \text{ in } N \mid V W \]
  \[ \mid \text{if } V \text{ then } M \text{ else } N \mid \text{read } \ell \mid \text{write}(\ell, V) \]
Selected typing rules

\[
\begin{align*}
\Gamma \vdash V_1 : A & \quad \Gamma \vdash V_2 : B & \quad \Gamma \vdash V : A_1 \times A_2 \\
\Gamma \vdash (V_1, V_2) : A \times B & & \Gamma \vdash \pi_i V : A_i
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A & \vdash M : TB \\
\Gamma \vdash \lambda x : A. M : A \rightarrow TB & \\
\Gamma \vdash V : A & \quad \Gamma \vdash M : TA & \quad \Gamma, x : A \vdash N : TB \\
\Gamma \vdash \text{val} \, V : TA & \quad \Gamma \vdash \text{let} \, x \leftarrow M \, \text{in} \, N : TB \\
\Gamma \vdash V : \text{bool} & \quad \Gamma \vdash M : TA & \quad \Gamma \vdash N : TA \\
& \quad \Gamma \vdash \text{if} \, V \, \text{then} \, M \, \text{else} \, N : TA
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{read}(\ell) : T\text{int} & \quad \Gamma \vdash \text{write}(\ell, V) : T\text{unit}
\end{align*}
\]
Base semantics

- Our language is simple: no recursion, no recursive types, global store of integer locations
- Obvious semantics in sets and functions:

\[
\begin{align*}
S & = \text{Locs} \rightarrow \mathbb{Z} \\
\text{[unit]} & = 1 \\
\text{[int]} & = \mathbb{Z} \\
\text{[bool]} & = \mathbb{B} \\
\text{[A \times B]} & = [A] \times [B] \\
\text{[A \rightarrow TB]} & = [A] \rightarrow [TB] \\
\text{[TA]} & = S \rightarrow S \times [A]
\end{align*}
\]
Refined types and subtyping

- **Types**

  \[
  X, Y ::= \text{unit} | \text{int} | \text{bool} | X \times Y | X \rightarrow T_\varepsilon Y
  \]

  \[
  \Theta ::= x_1 : X_1, \ldots, x_n : X_n
  \]

  \[
  \varepsilon \subseteq \bigcup_{\ell \in \mathcal{L}} \{ r_\ell, w_\ell \}
  \]

- **Subtyping**

  \[
  \frac{X \leq X}{X \leq X} \quad \frac{X \leq Y \quad Y \leq Z}{X \leq Z} \quad \frac{X \leq X' \quad Y \leq Y'}{X \times Y \leq X' \times Y'}
  \]

  \[
  \frac{X' \leq X \quad T_\varepsilon Y \leq T_\varepsilon' Y'}{(X \rightarrow T_\varepsilon Y) \leq (X' \rightarrow T_\varepsilon' Y')}
  \]

  \[
  \frac{\varepsilon \subseteq \varepsilon' \quad X \leq X'}{T_\varepsilon X \leq T_\varepsilon' X'}
  \]
Selected typing rules for refined types

\[ \Theta, x : X \vdash M : T_\varepsilon Y \]
\[ \Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon Y \]
\[ \Theta \vdash V_1 : X \to T_\varepsilon Y \quad \Theta \vdash V_2 : X \]
\[ \Theta \vdash V_1 V_2 : T_\varepsilon Y \]

\[ \Theta \vdash V : X \]
\[ \Theta \vdash \text{val } V : T_0 X \]
\[ \Theta \vdash M : T_\varepsilon X \quad \Theta, x : X \vdash N : T_\varepsilon Y \]
\[ \Theta \vdash \text{let } x \leftarrow M \text{ in } N : T_\varepsilon \cup \varepsilon Y \]
\[ \Theta \vdash V : \text{bool} \]
\[ \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon X \]
\[ \Theta \vdash \text{if } V \text{ then } M \text{ else } N : T_\varepsilon X \]

\[ \Theta \vdash \text{read}(\ell) : T_{\{r_\ell\}}(\text{int}) \]
\[ \Theta \vdash V : \text{int} \]
\[ \Theta \vdash \text{write}(\ell, V) : T_{\{w_\ell\}}(\text{unit}) \]
\[ \Theta \vdash V : X \quad X \leq X' \]
\[ \Theta \vdash V : X' \]
\[ \Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \leq T_\varepsilon' X' \]
\[ \Theta \vdash M : T_\varepsilon' X' \]
Erasure

- Define a map $U$ from refined type to underlying base type.
  
  $U(\text{int}) = \text{int}$
  $U(\text{bool}) = \text{bool}$
  $U(\text{unit}) = \text{unit}$
  
  $U(X \times Y) = U(X) \times U(Y)$
  $U(X \to T_\varepsilon Y) = U(X) \to U(T_\varepsilon Y)$
  $U(T_\varepsilon X) = T(U(X))$

- Easy results:
  
  If $X \leq Y$ then $U(X) = U(Y)$
  If $\Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$
Embedding base into refined

- Define map $G$ from base types to refined types that adds the “top” annotation to computation types:

  $G(\text{int}) = \text{int}$
  $G(\text{bool}) = \text{bool}$
  $G(\text{unit}) = \text{unit}$
  $G(A \times B) = G(A) \times G(B)$
  $G(A \rightarrow TB) = G(A) \rightarrow T_{\{r,w\}}G(B)$

- Easy result:
  
  If $\Gamma \vdash V : A$ then $G(\Gamma) \vdash V : G(A)$
Goal

- Use the semantics to validate equivalences at particular types. E.g. effect-independent:

\[
\begin{align*}
\Theta \vdash M : T_{\varepsilon_1} Y & \quad \Theta, y : Y \vdash N : T_{\varepsilon_2} X & \quad \Theta, x : X \vdash P : T_{\varepsilon_3} Z \\
\Theta \vdash \text{let } x \leftarrow (\text{let } y \leftarrow M \text{ in } N) \text{ in } P & \quad = \text{let } y \leftarrow M \text{ in let } x \leftarrow N \text{ in } P : T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} Z
\end{align*}
\]

- E.g. effect-dependent:

\[
\begin{align*}
\Theta \vdash M : T_{\varepsilon} X & \quad \Theta, x : X, y : X \vdash N : T_{\varepsilon'} Y \\
\Theta \vdash \text{let } x \leftarrow M \text{ in let } y \leftarrow M \text{ in } N & \quad = \text{let } x \leftarrow M \text{ in } N[x/y] : T_{\varepsilon \cup \varepsilon'} Y
\end{align*}
\]
Relational semantics of effects

- For a simple type $A$, the meaning $\llbracket A \rrbracket$ was a set.
- For a refined type $X$, the meaning $\llbracket X \rrbracket$ is a subset of the underlying set $\llbracket U(X) \rrbracket$ together with a coarser notion of equality.
- That is: a partial equivalence relation on $\llbracket U(X) \rrbracket$ (a symmetric, transitive relation).
- Roughly speaking:
  - $(x,x)$ in $\llbracket X \rrbracket$ = “semantically, $x$ behaves as type $X$”
  - $(x,y)$ in $\llbracket X \rrbracket$ = “semantically, $x$ is equivalent to $y$ at type $X$”
Semantics of refined types, on one slide

\[ [X] \subseteq [U(X)] \times [U(X)] \]
\[ \text{int} = \Delta_{\mathbb{Z}} \]
\[ \text{bool} = \Delta_{\mathbb{B}} \]
\[ \text{unit} = \Delta_{1} \]
\[ [X \times Y] = [X] \times [Y] \]
\[ [X \rightarrow T_{\varepsilon} Y] = [X] \rightarrow [T_{\varepsilon} Y] \]
\[ [T_{\varepsilon} X] = \bigcup_{R \in \mathcal{R}_{\varepsilon}} R \rightarrow R \times [X] \]

Values of base type are related just to themselves (diagonal relation)

Functions are related in the usual “logical” fashion: related arguments → related results

Computations are related if they preserve all state relations that respect the effect

\[ \mathcal{R}_{\varepsilon}, \mathcal{R}_{e} \subseteq \mathbb{P}(S \times S) \]
\[ \mathcal{R}_{\varepsilon} = \bigcap_{e \in \varepsilon} \mathcal{R}_{e} \]
\[ \mathcal{R}_{\times \ell} = \{R \mid \forall (s, s') \in R, s \ell = s' \ell\} \]
\[ \mathcal{R}_{w \ell} = \{R \mid \forall (s, s') \in R, \forall n \in \mathbb{Z}. (s[\ell \mapsto n], s'[\ell \mapsto n]) \in R\} \]
Effect-respecting relations

\[ [T_\varepsilon X] = \bigcap_{R \in \mathcal{R}_\varepsilon} R \to R \times [X] \]

- Computations are related if they produce related results and preserve all state relations \( R \) that respect \( \varepsilon \).”

- We say \( R \) respects reads from \( \ell \) if
  \[(s, s') \in R \Rightarrow s(\ell) = s'(\ell) \]

- We say \( R \) respects writes to \( \ell \) if
  \[(s, s') \in R \Rightarrow \forall x \in \mathbb{Z}. (s[\ell := x], s'[\ell := x]) \in R \]

- We say \( R \) respects \( \varepsilon \) if it respects each effect \( e \) in \( \varepsilon \) \( R \in \mathcal{R}_\varepsilon = \bigcap_{e \in \varepsilon} \mathcal{R}_e \)
Results

- **Soundness of subtyping:** If $X \leq Y$ then $[X] \subseteq [Y]$.

- **Fundamental theorem:**
  
  If $\Theta \vdash V : X, (\rho, \rho') \in [\Theta]$ then $(\llbracket U(\Theta) \vdash V : U(X) \rrbracket \rho, \llbracket U(\Theta) \vdash V : U(X) \rrbracket \rho') \in [X]$.

- **Meaning of top effect:** $[G(A)] = \Delta_{[A]}$.

- **Equivalences**
  - Effect-independent: congruence rules, $\beta$, $\eta$ rules, commuting conversions
  - Effect-dependent: dead computation, duplicated computation, commuting computations, pure lambda hoist
  - Reasoning is quite intricate, involving construction of specific effect-respecting relations. See paper!
Effect-dependent equivalences (1)

Dead Computation:

\[ \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon Y \]
\[ \Theta \vdash \text{let } x \leftarrow M \text{ in } N = N : T_\varepsilon Y \quad x \not\in \Theta, \text{wrs}(\varepsilon) = \emptyset \]

Duplicated Computation:

\[ \Theta \vdash M : T_\varepsilon X \quad \Theta, x : X, y : X \vdash N : T_\varepsilon Y \]
\[ \Theta \vdash \text{let } x \leftarrow M \text{ in let } y \leftarrow M \text{ in } N = \text{let } x \leftarrow M \text{ in } N[x/y] : T_{\varepsilon \cup \varepsilon}, Y \quad \text{rds}(\varepsilon) \cap \text{wrs}(\varepsilon) = \emptyset \]
Effect-dependent equivalences (2)

Commuting Computations:

\[
\begin{align*}
\Theta &\vdash M_1: T_{\varepsilon_1}X_1 \quad \Theta &\vdash M_2: T_{\varepsilon_2}X_2 \\
\Theta \vdash \text{let } x_1 \leftarrow M_1 \text{ in let } x_2 \leftarrow M_2 \text{ in } N &\vdash N: T_{\varepsilon_1}Y \\
\Theta \vdash \text{let } x_2 \leftarrow M_2 \text{ in let } x_1 \leftarrow M_1 \text{ in } N &\vdash N: T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_1}Y
\end{align*}
\]

\[\text{rds}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \emptyset\]
\[\text{wrs}(\varepsilon_1) \cap \text{rds}(\varepsilon_2) = \emptyset\]
\[\text{wrs}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \emptyset\]

Pure Lambda Hoist:

\[
\begin{align*}
\Theta &\vdash M: T_\{\}Z \quad \Theta, x : X, y : Z \vdash N: T_{\varepsilon}Y \\
\Theta &\vdash \text{val } (\lambda x : U(X). \text{let } y \leftarrow M \text{ in } N) \vdash \text{let } y \leftarrow M \text{ in } \text{val } (\lambda x : U(X). N) &\vdash T_\{\}(X \rightarrow T_{\varepsilon}Y)
\end{align*}
\]

This “currying” isomorphism follows from lambda-hoist and \(\beta\)-\(\eta\) rules:

\[
X \times Y \to T_{\varepsilon}Z \cong X \to T_{\emptyset}(Y \to T_{\varepsilon}Z)
\]
Other applications

- General approach – defining semantics of refined types by relations over semantics of base types – transfers to other effects.
  - Here, the effect is directly observable and the definition of relations is obvious. Nevertheless, they’re necessary: a predicate-based semantics is unable to validate non-trivial equivalences.

\[
[T \cdot A] = ([A] + E)_\bot
\]

\[
[\Delta \vdash T_\varepsilon X < T \cdot A]_\eta =
\{(\text{inl } m, \text{inl } m') \mid (m, m') \in [\Delta \vdash X < A]_\eta\}
\cup \{(\text{inr } E, \text{inr } E') \mid E \in [\Delta \vdash \varepsilon]_\eta\}
\cup \{(\bot, \bot) \mid \bot \in [\Delta \vdash \varepsilon]_\eta\}
\]
Further directions

- **Recursion**
  - Relations over domains instead of sets, unproblematic in absence of dynamic allocation

- **Dynamic allocation and regions**
  - runST-style encapsulation + its semantic analogue (e.g. purity of memoize function)
  - See PPDP’07 paper

- **Higher-typed store**
  - Much more challenging – work in progress
Dynamic Allocation and Local State

Joint work with Benjamin Leperchey
Build on

- Pitts and Stark
- Reddy and Yang
- Shinwell and Pitts
- (Meyer, Sieber, O’Hearn, Reynolds, Tennent, Oles, …)
Types and terms

\[ \tau ::= \text{unit} \mid \text{int} \mid \sigma \text{ ref} \mid \tau \times \tau \mid \tau + \tau \mid \tau \rightarrow T\tau \]

\[ \sigma ::= \text{int} \mid \sigma \text{ ref} \]

\[ \gamma ::= \tau \mid T\tau \]

\[ V ::= x \mid n \mid \ell \mid () \mid (V, V') \mid \text{in}_i^\tau V \mid \text{rec } f(x: \tau): \tau' = M \]

\[ M ::= V V' \mid \text{let } x \leftarrow M \text{ in } M' \mid \text{val } V \mid \pi_i V \mid \text{ref } V \mid !V \]

\[ V ::= V' \mid \text{case } V \text{ of } \text{in}_1 x \Rightarrow M ; \text{in}_2 x \Rightarrow M' \]

\[ \mid V = V' \mid V + V' \mid \text{iszero } V \]
Type Rules

\[
\begin{align*}
\text{(rec)} & \quad \Delta; \Gamma, x : \tau, f : \tau \to T(\tau') \vdash M : T(\tau') \\
\Delta; \Gamma \vdash (\text{rec} \ f(x : \tau) : \tau' = M) : \tau \to T(\tau') \\
\text{(loc)} & \quad \ell : \sigma \in \Delta \\
\Delta; \Gamma \vdash \ell : \sigma \text{ ref} \\
\text{(app)} & \quad \Delta; \Gamma \vdash V_1 : \tau \to T\tau' \\
\Delta; \Gamma \vdash V_2 : \tau \\
\Delta; \Gamma \vdash V_1 \ V_2 : T\tau' \\
\text{(let)} & \quad \Delta; \Gamma \vdash M_1 : T(\tau_1) \\
\Delta; \Gamma, x : \tau_1 \vdash M_2 : T(\tau_2) \\
\Delta; \Gamma \vdash \text{let} \ x \leftarrow M_1 \ \text{in} \ M_2 : T(\tau_2) \\
\text{(val)} & \quad \Delta; \Gamma \vdash V : \tau \\
\text{(eq)} & \quad \Delta; \Gamma \vdash V_1 : \sigma \text{ ref} \\
\Delta; \Gamma \vdash V_2 : \sigma \text{ ref} \\
\Delta; \Gamma \vdash V_1 = V_2 : T(\text{unit + unit}) \\
\text{(deref)} & \quad \Delta; \Gamma \vdash V : \sigma \text{ ref} \\
\Delta; \Gamma \vdash !V : T\sigma \\
\text{(alloc)} & \quad \Delta; \Gamma \vdash V : \sigma \\
\Delta; \Gamma \vdash \text{ref} \ V : T(\sigma \text{ ref}) \\
\text{(assign)} & \quad \Delta; \Gamma \vdash V_1 : \sigma \text{ ref} \\
\Delta; \Gamma \vdash V_2 : \sigma \\
\Delta; \Gamma \vdash V_1 := V_2 : T(\text{unit})
\end{align*}
\]
Continuation terms

\[
\Delta; \vdash \text{val } x : (x : \tau)^	op
\]

\[
\Delta; x : \tau \vdash M : T\tau' \quad \Delta; \vdash K : (y : \tau')^	op
\]

\[
\Delta; \vdash \text{let } y \leftarrow M \text{ in } K : (x : \tau)^	op
\]
\[
\begin{align*}
\Sigma, \text{let } x \leftarrow \text{val } V \text{ in } \text{val } x & \Downarrow \\
\Sigma, \text{let } x_2 \leftarrow M_1 \text{ in } (\text{let } x_1 \leftarrow M_2 \text{ in } K) & \Downarrow \\
\Sigma, \text{let } x_1 \leftarrow (\text{let } x_2 \leftarrow M_1 \text{ in } M_2) \text{ in } K & \Downarrow \\
\Sigma, \text{let } x_1 \leftarrow M[V/x_2], (\text{rec } f(x_2 : \tau_1 : \tau_2 = M) / f) \text{ in } K & \Downarrow \\
\Sigma, \text{let } x_1 \leftarrow (\text{rec } f(x_2 : \tau_1 : \tau_2 = M) V \text{ in } K & \Downarrow \\
\Sigma, \text{let } x \leftarrow \text{val } \text{false} \text{ in } K & \Downarrow \\
\Sigma, \text{let } x \leftarrow \ell = \ell' \text{ in } K & \Downarrow \\
\Sigma[\ell \mapsto \text{in}_{\mathbb{L}} \ell'], \text{let } x \leftarrow \text{val } () \text{ in } K & \Downarrow \\
\Sigma[\ell \mapsto \text{in}_{\mathbb{L}} \ell'], \text{let } x \leftarrow \ell := \ell' \text{ in } K & \Downarrow \\
\Sigma(\ell) = \text{in}_{\mathbb{L}} \ell' & \Sigma, \text{let } x \leftarrow \text{val } \ell' \text{ in } K & \Downarrow \\
\Sigma, \text{let } x \leftarrow !\ell \text{ in } K & \Downarrow \\
\Sigma[\ell \mapsto \text{in}_{\mathbb{L}} \ell'], \text{let } x \leftarrow \text{val } \ell \text{ in } K & \Downarrow \\
\Sigma[\ell \mapsto \text{in}_{\mathbb{L}} \ell'], \text{let } x \leftarrow \text{val } \ell \text{ in } K & \Downarrow \\
\Sigma, \text{let } x \leftarrow \text{ref } \ell' \text{ in } K & \Downarrow \\
\ell \not\in \text{locs}(\Sigma) \cup \text{locs}(K) \cup \{\ell'\}
\end{align*}
\]
let val r = ref 0 in
  fn () => r := !r+1; !r
end

let val r1 = ref 0 in
  fn () => r := !r-1; -(!r)
end
let val r = ref 0
    val s = ref 1
  in M

let val s = ref 1
  val r = ref 0
  in M
FM cpos

- Pitts, Gabbay, Shinwell
- do this in the board…
Semantics of types

\[ S = L \Rightarrow (\mathbb{Z} + L) \]

\[
\begin{align*}
\llbracket \text{unit} \rrbracket &= 1 \\
\llbracket \tau_1 \times \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\
\llbracket \text{int} \rrbracket &= \mathbb{Z} \\
\llbracket \tau_1 + \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket \\
\llbracket \sigma \text{ ref} \rrbracket &= L \\
\llbracket \tau_1 \rightarrow T\tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \Rightarrow T\llbracket \tau_2 \rrbracket \\
TD &= (S \Rightarrow D \Rightarrow \emptyset) \rightarrow (S \Rightarrow \emptyset)
\end{align*}
\]
Semantics of terms

\[ [\Delta; \Gamma \vdash \ell : \sigma \text{ ref}] \rho = \ell \]

\[ [\Delta; \Gamma \vdash \text{ let } x \leftarrow M_1 \text{ in } M_2 : T\tau_2] \rho \ k \ S = \]
\[ [\Delta; \Gamma \vdash M_1 : T\tau_1] \rho \ (\lambda S' : S.\lambda d : [\tau_1].\Delta; \Gamma, x : \tau_1 \vdash M_2 : T\tau_2] \rho[x \mapsto d] k \ S' \] \ S

\[ [\Delta; \Gamma \vdash \text{ val } V : T\tau] \rho \ k \ S = k \ S (\Delta; \Gamma \vdash V : \tau] \rho) \]

\[ [\Delta; \Gamma \vdash !V : T\sigma] \rho \ k \ S = \begin{cases} 
  k \ S \ v & \text{if } S([\Delta; \Gamma \vdash V : \sigma \text{ ref}] \rho) = \text{in}_{[\sigma]}v \\
  \bot & \text{otherwise} 
\end{cases} \]

\[ [\Delta; \Gamma \vdash V_1 := V_2 : T\text{ unit}] \rho \ k \ S = \]
\[ k \ S ([\Delta; \Gamma \vdash V_1 : \sigma \text{ ref}] \rho) \leftrightarrow \text{in}_{[\sigma]}([\Delta; \Gamma \vdash V_2 : \sigma] \rho)] * \]

\[ [\Delta; \Gamma \vdash (\text{ rec } f \ x = M) : \tau \rightarrow T\tau'] \rho = \]
\[ \text{fix}(\lambda f' : [\tau \rightarrow T\tau'].\lambda x' : [\tau].\Delta; \Gamma, f : \tau \rightarrow T\tau', x : \tau \vdash M : T\tau'] \rho[f \mapsto f', x \mapsto x']) \]
Soundness and Adequacy

If $\Delta; \vdash M : T_\tau$, $\Delta; \vdash K : (x : \tau)^\top$, $\Sigma : \Delta$ and $S \in \llbracket \Sigma \rrbracket$ then

$\Sigma$, let $x \leftarrow M$ in $K \downarrow$

$\iff$

$\llbracket \Delta; \vdash M : T_\tau \rrbracket \{\} \llbracket \Delta; \vdash K : (x : \tau)^\top \rrbracket^K S = \top$

If

$\llbracket \Delta; \Gamma \vdash G_1 : \gamma \rrbracket = \llbracket \Delta; \Gamma \vdash G_2 : \gamma \rrbracket$

then

$\Delta; \Gamma \vdash G_1 =_{ctx} G_2 : \gamma$
Semantic (in)equivalences

\[ \Delta; \Gamma \vdash V_1 : \sigma_1 \quad \Delta; \Gamma \vdash V_2 : \sigma_2 \quad \Delta; \Gamma, x : \sigma_1 \text{ ref, } y : \sigma_2 \text{ ref} \vdash N : T_\tau \]

\[ \Delta; \Gamma \vdash \text{ let } x \leftarrow \text{ ref } V_1 \text{ in (let } y \leftarrow \text{ ref } V_2 \text{ in } N) \quad =_{\text{ctx}} \quad \text{ let } y \leftarrow \text{ ref } V_2 \text{ in (let } x \leftarrow \text{ ref } V_1 \text{ in } N) : T_\tau \]

\[ \Delta; \Gamma \vdash V : \sigma \quad \Delta; \Gamma \vdash N : T_\tau \]

\[ \Delta; \Gamma \vdash \text{ let } x \leftarrow \text{ ref } V \text{ in } N \quad =_{\text{ctx}} \quad N : T_\tau \quad x \not\in f\nu N \]

\[ \exists \ell \in \mathbb{L}. S(\ell) = \text{ in}_{\mathbb{Z}}(3) \]
A Parametric Logical Relation

- Partially ordered set of parameters $p$
- Parameter-indexed relations
  \[ \forall p. \mathcal{R}_\mathcal{S}(p) \subseteq \mathcal{S} \times \mathcal{S} \]

\[ \forall p. \forall \gamma. \mathcal{R}_\gamma(p) \subseteq [\gamma] \times [\gamma] \]

- Show denotation of each term related to itself
- Corollary: terms with related denotations are contextually equivalent
Accessibility maps

- Support turns out not to help in defining “the part of the store about which a relation depends”:

\[ \{(S_1, S_2) \mid \exists \ell, S_1 \ell = 0 = S_2 \ell \} \]

An accessibility map \( A \) is a function from \( S \) to finite subsets of \( \mathbb{L} \), such that:

\[ \forall S, S' \in S, (\forall \ell \in AS, S\ell = S'\ell) \implies A(S) = A(S') \]

The subtyping ordering \( \prec \) is defined as:

\[ A \prec A' \iff \forall S, A(S) \supseteq A'(S) \]
Accessibility maps from state types

If $\Delta$ is a state type, then $\text{Acc}_\Delta : S \rightarrow P_{fin}(L)$ is defined by $\text{Acc}_\Delta(S) = \bigcup_{(\ell : \sigma) \in \Delta} \text{Acc}(\ell, \sigma, S)$ where $\text{Acc}(\ell, \text{int}, S) \overset{def}{=} \{\ell\}$ and

$$\text{Acc}(\ell, \text{ref}, S) \overset{def}{=} \{\ell\} \bigcup \begin{cases} \text{Acc}(\ell', \sigma, S) & \text{if } S\ell = \text{in}_{L\ell'} \\ \emptyset & \text{otherwise} \end{cases}$$

If $A$ is an accessibility map, we define $S \sim S' : A$ to mean $\forall \ell \in A(S), S\ell = S'\ell$. 
Finitary state relations

A finitary state relation $r$ is a pair $\langle |r|, A_r \rangle$ where $|r| \subseteq S \times S$ and $A_r$ is an accessibility map, subject to the following saturation condition: if $S_1 \sim S'_1 : A_r$ and $S_2 \sim S'_2 : A_r$ then $(S_1, S_2) \in |r| \iff (S'_1, S'_2) \in |r|$. 

Given two finitary state relations, $r_1 = \langle |r^1|, A^1 \rangle$ and $r_2 = \langle |r^2|, A^2 \rangle$, define

$$r^1 \otimes r^2 \overset{def}{=} \langle |r^1 \otimes r^2|, A^1 \land A^2 \rangle$$

where

$$(S_1, S_2) \in |r^1 \otimes r^2| \iff \left\{ \begin{array}{l} (S_1, S_2) \in |r^1| \cap |r^2| \\ \forall i \in \{1, 2\}, A^1(S_i) \cap A^2(S_i) = \emptyset \end{array} \right.$$
Parameters

A parameter is a pair \((\Delta, r)\), where \(\Delta\) is a state type and \(r\) is a finitary relation; we will abbreviate this to \(\Delta r\). If \(\Delta r\) is a parameter, we define the binary relation on states \(\mathcal{R}_S(\Delta r) \overset{\text{def}}{=} |id_\Delta \otimes r|\) and define the partial order \(\triangleright\) on parameters by

\[
\Delta r \triangleright \Delta' r' \iff (\Delta \supseteq \Delta') \land (\exists r'', r = r' \otimes r'')
\]
Logical Relation

\[
\begin{align*}
R_{\text{unit}}(\Delta r) &= \{(*, *)\} \\
R_{\text{int}}(\Delta r) &= \{(n, n) \mid n \in N\} \\
R_{\sigma_{\text{ref}}}(\Delta r) &= \{(\ell, \ell) \mid (\ell : \sigma) \in \Delta\} \\
R_{\tau \rightarrow T_{\tau'}}(\Delta r) &= \{(f_1, f_2) \mid \forall \Delta' r' \triangleright \Delta r, (v_1, v_2) \in R_{\tau}(\Delta' r'), (f_1v_1, f_2, v_2) \in R_{T_{\tau'}}(\Delta' r')\}
\end{align*}
\]

For continuations, we define \(R_{\tau \top}(\Delta r)\) to be

\[
\{(k_1, k_2) \mid \forall \Delta' r' \triangleright \Delta r, (v_1, v_2) \in R_{\tau}(\Delta' r'), (S_1, S_2) \in R_{\Sigma}(\Delta' r'), k_1S_1v_1 = k_2S_2v_2\}
\]

and for computations, \(R_{T_{\tau}}(\Delta r)\) is defined as

\[
\{(f_1, f_2) \mid \forall \Delta' r' \triangleright \Delta r, (k_1, k_2) \in R_{\tau \top}(\Delta' r'), (S_1, S_2) \in R_{\Sigma}(\Delta' r'), f_1k_1S_1 = f_2k_2S_2\}
\]
Why?

- **Fundamental Lemma:**
  If $\Delta; \Gamma \vdash G: \gamma$, then

  \[
  \forall r. (\llbracket \Delta; \Gamma \vdash G: \gamma \rrbracket, \llbracket \Delta; \Gamma \vdash G: \gamma \rrbracket) \in \mathcal{R}_{\Gamma \vdash \gamma}(\Delta r).
  \]

- **Soundness of relational reasoning:**
  If $\Delta; \Gamma \vdash G_i : \gamma$ for $i = 1, 2$ and

  \[
  (\llbracket \Delta; \Gamma \vdash G_1: \gamma \rrbracket, \llbracket \Delta; \Gamma \vdash G_2: \gamma \rrbracket) \in \mathcal{R}_{\Gamma \vdash \text{ctx} \Gamma}(\Delta \top)
  \]

  then $\Delta; \Gamma \vdash G_1 =_{\text{ctx}} G_2 : \gamma$. 
Examples

- The garbage collection rule from earlier
- All the Meyer-Sieber examples, e.g.

```plaintext
let x ← ref 0 in
    let almost_add2 ← λz. if z = x
        then x := 1
        else let y ← !x in let y' ← y + 2 in x := y'
in
    p(almost_add2);
    let y ← !x in
        if !x mod 2 = 0 then diverge_unit else val ()
```

always diverges.
Examples

- Pointers between hidden and visible parts:
  
  ```
  let x ← \text{ref 0} \text{ in}
  let y ← \text{ref } x \text{ in}
  p \ x;
  let z ← !y \text{ in}
  if \ z = x \text{ then } \text{dverge}_{\text{unit}} \text{ else \ val } ()
  ```

- Some very artificial encodings of crypto a la Sumii and Pierce
Non-examples 😞

$$M = \text{let } x \leftarrow \text{ref } 0 \text{ in}$$
$$\quad \quad p(\lambda_. x := 1; 0);$$
$$\quad \quad \text{let } y \leftarrow !x \text{ in}$$
$$\quad \quad \quad \text{if iszero } y \text{ then } \text{val } () \text{ else } \text{diverge}_{\text{unit}}$$

$$N = \quad p\left(\lambda_. \text{diverge}_{\text{int}}\right)$$

snapback $f \ k \ S = f \ast \left(\lambda S'. \lambda n. k \ S \ n\right) S$
Further work

- See Lars Birkedal and Nina Bohr, “Relational Reasoning for Recursive Types and References” APLAS’06 for extension to higher-typed store and recursive types.
- Nina’s thesis goes further – parametric polymorphism and refinements to parameters to rule out snapback etc.