Rely-Guarantee Protocols

Filipe Militão\textsuperscript{1,2}  Jonathan Aldrich\textsuperscript{1}  Luís Caires\textsuperscript{2}

\textsuperscript{1}Carnegie Mellon University, Pittsburgh, USA
\textsuperscript{2}Universidade Nova de Lisboa, Lisboa, Portugal
Motivation

- Mutable state can be useful in certain cases.
- Precisely tracking the properties of mutable state avoids a class of state-related errors.
- However, aliasing makes tracking such properties challenging.
x.open(...);
x.write(...);
x.write(...);
Opened

```python
x.open(...);
x.write(...);
x.write(...);
```
\texttt{x.open(...);}
\texttt{x.write(...);}
\texttt{\textcolor{black}{y.close()}\textcolor{green}{\texttt{\textcolor{black}{x.write(...)}\textcolor{black}{x.write(...);}}}}
y.close() → Closed

Closed → x.open(...);

Closed → x.write(...);

Closed → x.write(...);
The assumption that \( x \) was pointing to an Opened file can be invalidated due to the interference caused by \( y \).
Contribution

A novel interference-control mechanism, **Rely-Guarantee Protocols**, to statically handle interference in the use of mutable state that is shared by aliases through statically disconnected variables.
Language

- Polymorphic $\lambda$-calculus with mutable references (and immutable records, tagged sums, ...).

- Technically, we use a variant of $L^3$ adapted for usability and extended with new constructs, and our sharing mechanism.

State as a Linear Resource

ref A
State as a Linear Resource

ref A

reference with contents of type A
State as a Linear Resource

ref $A$

reference with contents of type $A$

ref $p$

rw $p A$
State as a Linear Resource

duplicable reference to location $p$

reference with contents of type $A$
State as a Linear Resource

`ref A`  

Reference with contents of type `A`

duplicable reference to location `p`

`rw p A`
State as a Linear Resource

- Reference with contents of type A
- Duplicable reference to location p
- Linear ("unique") read+write capability of location p with contents of type A
State as a Linear Resource

reference with contents of type A

duplicable reference to location \( p \)

linear ("unique") read+write capability of location \( p \) with contents of type A

\[ \text{ref}^A \]
\texttt{x : ref } p \texttt{ rw } p \texttt{ string}

\begin{verbatim}
let \ y = \ x \ in \\
\ x \ := \ 1 \ ; \\
\ delete \ \ y \ ; \\
\ x \ := \ false \\
end
\end{verbatim}
let \( y = x \) in

\[
\begin{align*}
  y & : \text{ref } p \\
  x & : \text{ref } p \\
  \text{rw } p & \text{ string}
\end{align*}
\]

\[
  x := 1;
\]

\[
  \text{delete } y;
\]

\[
  x := \text{false}
\]

end
let \( y = x \) in
\[
x := 1;
\]
\[
y : \text{ref } p \quad x : \text{ref } p \quad \text{rw } p \text{ int}
\]
delete \( y \);
\[
x := \text{false}
\]
end
let $y = x$ in
  $x := 1$;
  delete $y$;

$y : \text{ref } p \quad x : \text{ref } p$

  $x := \text{false}$

end
let \( y = x \) in
  \( x := 1; \)
delete \( y; \)

\( \textcolor{red}{y : \text{ref } p} \quad \textcolor{green}{x : \text{ref } p} \)

\( \textcolor{blue}{x := \text{false}} \)

end

\textbf{Type Error:} Missing capability to location \textcolor{red}{p}.
Why sharing?

Capabilities are linear (a.k.a. “unique”)!
Why sharing?

Capabilities are *linear* (a.k.a. “unique”)!
Sharing

A capability is split into rely-guarantee protocols to safely coordinate access to the shared state.
Sharing

A capability is split into **rely-guarantee protocols** to safely coordinate access to the shared state.
Disjoint

• Linearity ensured disjointness.

• Sharing causes *fictional* disjointness.

[Dinsdale-Young, et al. Concurrent Abstract Predicates. (ECOOP’10), and other works].
Fictionally Disjoint

• Linearity ensured disjointness.

• Sharing causes *fictional* disjointness.

[Dinsdale-Young, et al. Concurrent Abstract Predicates. (ECOOP’10), and other works].
Problems of Sharing

1. Account for interference (*public* changes).
   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle *private* changes.
   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like *int* to *string*, etc.).
Alias Interleaving

```plaintext
x := 1;
doSomething();
!x // what do we get?
```
Alias Interleaving

fun().l

fun().x := false

fun().delete x

x := 1;
doSomething();

!x // what do we get?

doSomething interleave zero or more aliases to the same state as referenced by x.
If `doSomething` did change the same state as aliased by `x` (i.e. interfered), what change occurred?

```plaintext
x := 1;
doSomething();
!x  // what do we get?
```

If `doSomething` did change the same state as aliased by `x` (i.e. interfered), what change occurred?
Handling Interference

• One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

\[ \text{I} \Rightarrow \text{I} \]

• Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

\[ \text{R} \Rightarrow \text{G} \]

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.
Handling Interference

- One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

![](image.png)

- Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

![](image.png)

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.
Handling Interference

- One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

  \[ I \implies I \]

- Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

  \[ R \implies G \]

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.
Modeling Interference

A Rely-Guarantee Protocol models the shared state interaction from the alias’ own view/perspective:

- The alias’ actions are constrained to fit within what the protocol specifies/allows.
- Interference is observed through new state(s) that *may* appear when inspecting the shared state. Thus, the protocol may specify actions over states that can only be produced by other aliases.
Rely-Guarantee Protocols

• An interference-control mechanism.
• I will focus on presenting the following:
  1. Protocol Specification (“public changes”)
  2. Protocol Use (“private changes”)
  3. Protocol Conformance (“alias interleaving”)

(see the paper for more technical details)
Types

$A ::= !A$ (pure/persistent)
| $A \rightarrow A$ (linear function)
| $A :: A$ (stacking)
| $A \times A$ (separation)
| $[f : A]$ (record)
| $X$ (type variable)
| $\forall X.A$ (universal type quantification)
| $\exists X.A$ (existential type quantification)
| $\forall t.A$ (universal location quantification)
| $\exists t.A$ (existential location quantification)

| $\text{ref } p$ (reference type)
| $\text{rec } X.A$ (recursive type)
| $\sum_i 1_i#A_i$ (tagged sum)
| $A \oplus A$ (alternative)
| $A \& A$ (intersection)
| $\text{rw } p A$ (read-write capability to $p$)
| $\text{none}$ (empty capability)
| $A \Rightarrow A$ (rely)
| $A; A$ (guarantee)
# Types

<table>
<thead>
<tr>
<th>( A ::= )</th>
<th>( \text{ref } p )</th>
<th>(reference type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \rightarrow A )</td>
<td>( \text{rec } X.A )</td>
<td>(recursive type)</td>
</tr>
<tr>
<td>( A :: A )</td>
<td>( \Sigma_i 1_i # A_i )</td>
<td>(tagged sum)</td>
</tr>
<tr>
<td>( A * A )</td>
<td>( A \oplus A )</td>
<td>(alternative)</td>
</tr>
<tr>
<td>( [f : A] )</td>
<td>( A &amp; A )</td>
<td>(intersection)</td>
</tr>
<tr>
<td>( X )</td>
<td>( \text{rw } p A )</td>
<td>(read-write capability to ( p ))</td>
</tr>
<tr>
<td>( \forall X.A )</td>
<td>\text{none}</td>
<td>(empty capability)</td>
</tr>
<tr>
<td>( \exists X.A )</td>
<td>( A \Rightarrow A )</td>
<td>(rely)</td>
</tr>
<tr>
<td>( \forall t.A )</td>
<td>( A; A )</td>
<td>(guarantee)</td>
</tr>
<tr>
<td>( \exists t.A )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( !A \) (pure/persistent)
- \( A \rightarrow A \) (linear function)
- \( A :: A \) (stacking)
- \( A * A \) (separation)
- \( [f : A] \) (record)
- \( X \) (type variable)
- \( \forall X.A \) (universal type quantification)
- \( \exists X.A \) (existential type quantification)
- \( \forall t.A \) (universal location quantification)
- \( \exists t.A \) (existential location quantification)
### Types

- $A ::= !A$ (pure/persistent)
- $A \rightarrow A$ (linear function)
- $A : A$ (stacking)
- $A * A$ (separation)
- $[f : A]$ (record)
- $X$ (type variable)
- $\forall X.A$ (universal type quantification)
- $\exists X.A$ (existential type quantification)
- $\forall t.A$ (universal location quantification)
- $\exists t.A$ (existential location quantification)

<table>
<thead>
<tr>
<th>$\text{ref } p$</th>
<th>(reference type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rec } X.A$</td>
<td>(recursive type)</td>
</tr>
<tr>
<td>$\sum_i 1_i # A_i$</td>
<td>(tagged sum)</td>
</tr>
<tr>
<td>$A \oplus A$</td>
<td>(alternative)</td>
</tr>
<tr>
<td>$A &amp; A$</td>
<td>(intersection)</td>
</tr>
<tr>
<td>$\text{rw } p A$</td>
<td>(read-write capability to $p$)</td>
</tr>
<tr>
<td>none</td>
<td>(empty capability)</td>
</tr>
<tr>
<td>$A \Rightarrow A$</td>
<td>(rely)</td>
</tr>
<tr>
<td>$A; A$</td>
<td>(guarantee)</td>
</tr>
</tbody>
</table>
## Types

<table>
<thead>
<tr>
<th>$A ::= !A$ (pure/persistent)</th>
<th>$\text{ref } p$ (reference type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow A$ (linear function)</td>
<td>$\text{rec } X.A$ (recursive type)</td>
</tr>
<tr>
<td>$A :: A$ (stacking)</td>
<td>$\sum_{i} 1_{i}#A_{i}$ (tagged sum)</td>
</tr>
<tr>
<td>$A * A$ (separation)</td>
<td>$A \oplus A$ (alternative)</td>
</tr>
<tr>
<td>$[f : A]$ (record)</td>
<td>$A &amp; A$ (intersection)</td>
</tr>
<tr>
<td>$X$ (type variable)</td>
<td>$\text{rw } p A$ (read-write capability to $p$)</td>
</tr>
<tr>
<td>$\forall X.A$ (universal type quantification)</td>
<td>$\text{none}$ (empty capability)</td>
</tr>
<tr>
<td>$\exists X.A$ (existential type quantification)</td>
<td>$A \Rightarrow A$ (rely)</td>
</tr>
<tr>
<td>$\forall t.A$ (universal location quantification)</td>
<td>$A ; A$ (guarantee)</td>
</tr>
<tr>
<td>$\exists t.A$ (existential location quantification)</td>
<td></td>
</tr>
</tbody>
</table>
Types

\[ A ::= !A \quad \text{(pure/persistent)} \]
\[ A \rightarrow A \quad \text{(linear function)} \]
\[ A :: A \quad \text{(stacking)} \]
\[ A \times A \quad \text{(separation)} \]
\[ [f : A] \quad \text{(record)} \]
\[ X \quad \text{(type variable)} \]
\[ \forall X.A \quad \text{(universal type quantification)} \]
\[ \exists X.A \quad \text{(existential type quantification)} \]
\[ \forall t.A \quad \text{(universal location quantification)} \]
\[ \exists t.A \quad \text{(existential location quantification)} \]

| \quad \text{ref } p \quad \text{(reference type)} |
| \quad \text{rec } X.A \quad \text{(recursive type)} |
| \quad \sum \limits_{i} 1_i \# A_i \quad \text{(tagged sum)} |
| \quad A \oplus A \quad \text{(alternative)} |
| \quad A \& A \quad \text{(intersection)} |
| \quad \text{rw } p.A \quad \text{(read-write capability to } p) |
| \quad \text{none} \quad \text{(empty capability)} |
| \quad A \Rightarrow A \quad \text{(rely)} |
| \quad A ; A \quad \text{(guarantee)} |
Types

\[
A ::= \ !A \quad \text{(pure/persistent)} \quad | \quad \text{ref } p \quad \text{(reference type)}
\]

\[
A \leadsto A \quad \text{(linear function)} \quad | \quad \text{rec } X.A \quad \text{(recursive type)}
\]

\[
A :: A \quad \text{(stacking)} \quad | \quad \Sigma_i 1_i \#A_i \quad \text{(tagged sum)}
\]

\[
A * A \quad \text{(separation)} \quad | \quad A \oplus A \quad \text{(alternative)}
\]

\[
[f : A] \quad \text{(record)} \quad | \quad A \& A \quad \text{(intersection)}
\]

\[
X \quad \text{(type variable)} \quad | \quad \text{rw } p \ A \quad \text{(read-write capability to } p)\]

\[
\forall X.A \quad \text{(universal type quantification)} \quad | \quad \text{none} \quad \text{(empty capability)}
\]

\[
\exists X.A \quad \text{(existential type quantification)} \quad | \quad A \Rightarrow A \quad \text{(rely)}
\]

\[
\forall t.A \quad \text{(universal location quantification)} \quad | \quad A; A \quad \text{(guarantee)}
\]

\[
\exists t.A \quad \text{(existential location quantification)}
\]
I. Protocol Specification

\[ P ::= \text{rec } X.P \mid X \mid P \oplus P \mid P \& P \mid A \Rightarrow P \mid A; P \mid \text{none} \]
I. Protocol Specification

\[ P ::= \text{rec } X.P \mid X \mid P \oplus P \mid P \& P \mid A \Rightarrow P \mid A; P \mid \text{none} \]
I. Protocol Specification

\[ P ::= \text{rec } X.P \mid X \mid (P \oplus P) \mid P \& P \mid A \Rightarrow P \mid A; P \mid \text{none} \]
I. Protocol Specification

\[ P ::= \text{rec} \ X.P \mid X \mid P \oplus P \mid P \& P \mid A \Rightarrow P \mid A; P \mid \text{none} \]
The shared state satisfies A, and requires the alias to obey the guarantee P.
I. Protocol Specification

\[ P ::= \text{rec } X.P \mid X \mid P \oplus P \mid P \& P \mid A \Rightarrow P \mid [A; P] \mid \text{none} \]

Requires the client to establish (guarantee) that the shared state satisfies \( A \) before continuing the use of the protocol as \( P \).
I. Protocol Specification

\[ P ::= \text{rec } X.P \mid X \mid P \oplus P \mid P \& P \mid A \Rightarrow P \mid A; P \mid \text{none} \]
Shared Pipe

Shared by two aliases interacting via a common buffer, here modeled as a *singly linked list*.

1. The **Producer** alias may **put** new elements in or **close** the pipe.

2. The **Consumer** alias may only **tryTake** elements from the buffer.

The result of **tryTake** is one of the following states: either there was some **Result**, or **NoResult**, or the pipe is fully **Depleted**.
Pipe
Producer

Consumer
Producer Protocol

Shared Buffer

Consumer Protocol

Consumer

Producer
Producer

tail: Empty $\Rightarrow$ (Filled $\oplus$ Closed); none
Producer

tail: Empty $\Rightarrow$ (Filled $\oplus$ Closed); none
tail: none
Producer

tail: Empty ⇒ ( Filled ⊕ Closed ) ; none
Producer

tail: Empty ⇒ ( Filled + Closed ) ; none

head: rec X.( ( Empty ⇒ Empty ; X )
+ ( Filled ⇒ none ; none )
+ ( Closed ⇒ none ; none ) )
Producer

tail: Empty $\Rightarrow$ (Filled $\oplus$ Closed); none

Consumer

head: rec X.( (Empty $\Rightarrow$ Empty; X)

$\oplus$ (Filled $\Rightarrow$ none; none)

$\oplus$ (Closed $\Rightarrow$ none; none) )
tail: **Empty** ⇒ ( **Filled** ⊕ **Closed** ) ; none

head: **none**
Producer

tail: Empty \Rightarrow ( \text{Filled} \oplus \text{Closed} ) ; \text{none}

head : \text{rec } X. ( ( \text{Empty} \Rightarrow \text{Empty} ; X ) \oplus ( \text{Filled} \Rightarrow \text{none} ; \text{none} ) \oplus ( \text{Closed} \Rightarrow \text{none} ; \text{none} ) )

Consumer
Producer Protocol:

\[
\text{tail} : \text{Empty} \Rightarrow (\text{Filled} \oplus \text{Closed}) ; \text{none}
\]

Consumer Protocol:

\[
\text{head} : \text{rec} \ X. (\ (\text{Empty} \Rightarrow \text{Empty} ; X) \\
\oplus (\text{Filled} \Rightarrow \text{none} ; \text{none}) \\
\oplus (\text{Closed} \Rightarrow \text{none} ; \text{none}) )
\]
**Producer Protocol:**

\[ T[t] = \text{rw} \ t \ \text{Empty}[^\#[\ ] [\ ] ] \Rightarrow \]

\[ ( (\text{rw} \ t \ \text{Node}[^\#[\ ] [\ ] ] [\ ] [\ ] ] ) \oplus (\text{rw} \ t \ \text{Closed}[^\#[\ ] [\ ] ] [\ ] [\ ] ] ) ) ) ; \]

\[
\text{none}
\]

**Consumer Protocol:**

\[ H[t] = \text{rec } X. (\]

\[ ( \text{rw} \ t \ \text{Empty}[^\#[\ ] [\ ] ] [\ ] [\ ] ] \Rightarrow \text{rw} \ t \ \text{Empty}[^\#[\ ] [\ ] ] [\ ] [\ ] ] ; X ) \]

\[ \oplus ( \text{rw} \ t \ \text{Node}[^\#[\ ] [\ ] ] [\ ] [\ ] ] [\ ] [\ ] ] \Rightarrow \text{none} ; \text{none} ) \]

\[ \oplus ( \text{rw} \ t \ \text{Closed}[^\#[\ ] [\ ] ] [\ ] [\ ] ] [\ ] [\ ] ] \Rightarrow \text{none} ; \text{none} ) \) \]
Pipe Typestate

\[ \exists P \exists C. ( \exists ![ \begin{array}{l}
\text{put} : ![\begin{array}{l}
( \text{int} :: P ) \rightarrow ( ![ [] :: P ) \end{array}]
\end{array}),
\text{close} : ![\begin{array}{l}
( ![ [] :: P ) \rightarrow ![ [] \end{array}]
\end{array}),
\text{tryTake} : ![\begin{array}{l}
( ![ [] :: C ) \rightarrow \text{Depleted}![ [] +
\text{NoResult}![ [] :: C ) + \text{Result}![\begin{array}{l}
( \text{int} :: C ) \end{array}]
\end{array}]),
\end{array}]) :: ( C * P ) \) \]
Pipe Typestate

\[ \exists P, \exists C. \left( \begin{array}{l}
\text{put} : !(\ (\ !\text{int} :: P ) \to (\ ![] :: P ) ) , \\
\text{close} : !(\ (\ ![] :: P ) \to ![] ) , \\
\text{tryTake} : !(\ (\ ![] :: C ) \to \text{Depleted}#![] + \text{NoResult}#(![] :: C) + \text{Result}#(!\text{int} :: C) ) \\
\end{array} \right) :: (\ C \times P ) \]
$$\exists P. \exists C. \begin{array}{l}
\text{put} : \ ! ( ( \ ! \text{int} :: P ) \rightarrow ( \ ! [ ] :: P ) ) , \\
\text{close} : \ ! ( ( \ ! [ ] :: P ) \rightarrow ! [ ] ) , \\
\text{tryTake} : \ ! ( ( \ ! [ ] :: C ) \rightarrow \text{Depleted} # ! [ ] + \\
\text{NoResult} # ( ! [ ] :: C ) + \text{Result} # ( ! \text{int} :: C ) ) \\
\end{array} \\
\mathbb{C} :: ( C \times P )
$$

$$\text{rw } p \ \exists p. ( \text{ref } p :: T [ p ] )$$
∃P.∃C. (∀[ ]::P)

| put         | !( ( !int :: P ) ⊸ ( ![ ] :: P ) ) |
| close       | !( ( ![ ] :: P ) ⊸ ![ ] )          |
| tryTake     | !( ( ![ ] :: C ) ⊸ Depleted#![] + NoResult#(![] :: C) + Result#(!int :: C) ) |

] :: ( C * P )

rw p ∃p.(ref p :: T[p])

rw c ∃p.(ref p :: H[p])
Problems of Sharing

1. Account for interference (public changes).
   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle private changes.
   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like int to string, etc.).
Syntax

\[
\begin{align*}
  v & ::= \rho \quad \text{(address)} & v.f & \text{(field)} \\
  | \quad x & \quad \text{(variable)} & v \; v & \text{(application)} \\
  | \quad \text{fun}(x : A).e & \quad \text{(function)} & \text{let } x = e \text{ in } e \text{ end} & \text{(let)} \\
  | \quad \langle t \rangle e & \quad \text{(location abstraction)} & \text{open } \langle t, x \rangle = v \text{ in } e \text{ end} & \text{(open location)} \\
  | \quad \langle X \rangle e & \quad \text{(type abstraction)} & \text{open } \langle X, x \rangle = v \text{ in } e \text{ end} & \text{(open type)} \\
  | \quad \langle p, v \rangle & \quad \text{(pack location)} & \text{new } v & \text{(cell creation)} \\
  | \quad \langle A, v \rangle & \quad \text{(pack type)} & \text{delete } v & \text{(cell deletion)} \\
  | \quad \{ f = v \} & \quad \text{(record)} & !v & \text{(dereference)} \\
  | \quad 1#v & \quad \text{(tagged value)} & v := v & \text{(assign)} \\

  e & ::= v \quad \text{(value)} & \text{case } v \text{ of } 1#x \rightarrow e \text{ end} & \text{(case)} \\
  | \quad v[p] & \quad \text{(location application)} & \text{share } A_0 \text{ as } A_1 \parallel A_2 & \text{(share)} \\
  | \quad v[A] & \quad \text{(type application)} & \text{focus } A & \text{(focus)} \\
  & \quad \text{defocus} & \text{defocus} & \text{(defocus)}
\end{align*}
\]
Syntax

\[
\begin{align*}
  v & ::= \rho & \text{(address)} \\
  & | x & \text{(variable)} \\
  & | \text{fun}(x : A).e & \text{(function)} \\
  & | \langle t \rangle e & \text{(location abstraction)} \\
  & | \langle X \rangle e & \text{(type abstraction)} \\
  & | \langle p, v \rangle & \text{(pack location)} \\
  & | \langle A, v \rangle & \text{(pack type)} \\
  & | \{ f = v \} & \text{(record)} \\
  & | 1\#v & \text{(tagged value)} \\
  e & ::= v & \text{(value)} \\
  & | v[p] & \text{(location application)} \\
  & | v[A] & \text{(type application)} \\
  & | v.f & \text{(field)} \\
  & | v \; v & \text{(application)} \\
  & | \text{let } x = e \text{ in } e \text{ end} & \text{(let)} \\
  & | \text{open } \langle t, x \rangle = v \text{ in } e \text{ end} & \text{(open location)} \\
  & | \text{open } \langle X, x \rangle = v \text{ in } e \text{ end} & \text{(open type)} \\
  & | \text{new } v & \text{(cell creation)} \\
  & | \text{delete } v & \text{(cell deletion)} \\
  & | !v & \text{(dereference)} \\
  & | v := v & \text{(assign)} \\
  & | \text{case } v \text{ of } 1\#x \rightarrow e \text{ end} & \text{(case)} \\
  & | \text{share } A_0 \text{ as } A_1 || A_2 & \text{(share)} \\
  & | \text{focus } \overline{A} & \text{(focus)} \\
  & | \text{defocus} & \text{(defocus)}
\end{align*}
\]
Syntax

(v.1)

(v v)

let x = e in e end

(open \( \langle t, x \rangle = v \) in e end)

(open \( \langle X, x \rangle = v \) in e end)

new v

delete v

!v

v := v

case v of \( 1 \# x \rightarrow e \) end

share \( A_0 \) as \( A_1 \parallel A_2 \)

focus \( \overline{A} \)

defocus
2. Protocol Use

• Protocols are used through **focus** and **defocus** constructs.

• They serve two purposes:

  a) **Hide private changes** from the other aliases of that shared state.

  b) **Advance the step** of the protocol, by obeying the constraints on **public** changes.
Focus / Defocus

focus Empty

...
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

...
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

...
Focus / Defocus

focused state

Empty  ➔ Filled ; Next

focus  Empty

...  defocus

defocus-guarantee
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

... 

defocus
Focus / Defocus

Empty \Rightarrow \text{Filled} ; \text{Next}

focus \quad \text{Empty}

\text{Empty} , \quad \text{Filled} ; \text{Next}

defocus
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

Empty , Filled ; Next

PartiallyFilled , Filled ; Next

defocus
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

Empty , Filled ; Next

...}

Filled , Filled ; Next

defocus
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

Focus Empty

Filled , Filled ; Next

Defocus
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

Focus: Empty

Empty , Filled ; Next

...  

Filled , Filled ; Next

Defocus
Focus / Defocus

Empty $\Rightarrow$ Filled ; Next

focus Empty

Empty , Filled ; Next

... 

Filled , Filled ; Next

defocus

Next
Focus / Defocus

- **Empty** ⇒ **Filled** ; **Next** , ∆

**focus  Empty**

- **Empty** , **Filled** ; **Next** ▷ ∆

**private changes**

- ...

**defocus**

- **Filled** , **Filled** ; **Next** ▷ ∆

- **Next** , ∆
Focus / Defocus

private changes

focus Empty

Empty ⇒ Filled ; Next △

Empty, Filled ; Next △

...
Focus / Defocus

private changes

Focus

Empty \Rightarrow Filled ; Next , \Delta

Empty, Filled ; Next ▷ Δ

Filled, Filled; Next ▷ Δ

Defocus

Next , Δ

hides any state that may allow reentrant accesses to focused state
Problems of Sharing

1. Account for interference (*public* changes).
   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle *private* changes.
   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like `int` to `string`, etc.).
3. Protocol Conformance

• Protocols are introduced explicitly, in pairs, through the share construct:

\[ \text{share } A \text{ as } B \| | \ C \]

“type A (either a capability or an existing protocol) can be safely split in types B and C (two protocols)”

• Arbitrary aliasing is possible by continuing to split an existing protocol.
Checking share

- We must check that a protocol is aware of all possible states that may appear due to the “interleaving” of other aliases of that shared state.

- Checking a split is built from two components:
  a) a *stepping relation*, that “simulates” a single use of *focus-defocus* (i.e. a step of the protocol).
  b) a *protocol conformance definition* that ensures the protocol considers all possible alias interleaving.
share $E$ as

$\text{rec } X. ( E \Rightarrow E; X \oplus N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} )$

\[
\begin{array}{c}
\downarrow \\
E \Rightarrow ( N \oplus C )
\end{array}
\]
Protocol
Conformance
Example

share $E$ as

\[
\text{Consumer}
\]
\[\text{rec } X. ( E \Rightarrow E; X \oplus N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} )\]

\[\text{Producer}
\]
\[E \Rightarrow (N \oplus C)\]
Initial state.

possible interleaving
Possible interleaving

- Initial state.

- However, our protocols can only list a finite number of distinct states, and each protocol lists a finite number of distinct protocol steps.

  This will ensure that there is finite number of distinct configurations, each representing one possible alias interleaving in the use of the state that is being shared by the protocols.
State: \( E \)

**Consumer**

- \( E \Rightarrow E \)
- \( C \Rightarrow \text{none} \)
- \( N \Rightarrow \text{none} \)

**Producer**

- \( E \Rightarrow N \oplus C \)

**Configurations:**

\[
\langle E \Rightarrow \text{rec } X.(N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle
\]

\[
\langle N \oplus C \Rightarrow \text{rec } X.(N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel \text{none} \rangle
\]

\[
\langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle
\]
State:

Configurations:
State:

Consumer

\[
\begin{align*}
E & \implies E \\
C & \implies \text{none} \\
N & \implies \text{none}
\end{align*}
\]

Producer

\[
\begin{align*}
E & \implies N \oplus C
\end{align*}
\]

Configurations:

\[
\begin{align*}
\text{rec } & X. (N \implies \text{none} \oplus C \implies \text{none} \oplus E \implies E; X) \parallel E \implies (N \oplus C) \\
\text{rec } & X. (N \implies \text{none} \oplus C \implies \text{none} \oplus E \implies E; X) \parallel \text{none} \\
\text{none} & \implies \text{none} \parallel \text{none}
\end{align*}
\]
State: $E$

Configurations:

\[
\begin{align*}
\langle E \Rightarrow \text{rec } X \cdot (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle \\
\langle N \oplus C \Rightarrow \text{rec } X \cdot (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel \text{none} \rangle \\
\langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle
\end{align*}
\]
State: $E$

**Configurations:**

\[
\langle E \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle \\
\langle N \oplus C \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel \text{none} \rangle \\
\langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle
\]
Configurations:

\[
\langle E \Rightarrow \text{rec } X.(N \Rightarrow \text{none } \oplus C \Rightarrow \text{none } \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C)\rangle
\]

\[
\langle N \oplus C \Rightarrow \text{rec } X.(N \Rightarrow \text{none } \oplus C \Rightarrow \text{none } \oplus E \Rightarrow E; X) \parallel \text{none}\rangle
\]

\[
\langle \text{none } \Rightarrow \text{none } \parallel \text{none}\rangle
\]
State:

Configurations:

\[
\begin{align*}
\langle E \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle \\
\langle N \oplus C \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel \text{none} \rangle \\
\langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle
\end{align*}
\]
State:

Consumer

\[ \begin{align*}
E & \Rightarrow E \\
C & \Rightarrow \text{none} \\
N & \Rightarrow \text{none}
\end{align*} \]

Producer

\[ \begin{align*}
E & \Rightarrow \text{none} + C \\
N & \Rightarrow \text{none}
\end{align*} \]

Configurations:

\[ \langle E \Rightarrow \text{rec } X.(N \Rightarrow \text{none} + C \Rightarrow \text{none} + E \Rightarrow E; X) \parallel E \Rightarrow (N + C) \rangle \]
\[ \langle N + C \Rightarrow \text{rec } X.(N \Rightarrow \text{none} + C \Rightarrow \text{none} + E \Rightarrow E; X) \parallel \text{none} \rangle \]
\[ \langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle \]
State: \( N \oplus C \)

**Configurations:**

\[
\begin{align*}
\langle E \Rightarrow & \text{ rec } X. (N \Rightarrow \text{ none } \oplus C \Rightarrow \text{ none } \oplus E \Rightarrow E ; X) \parallel E \Rightarrow (N \oplus C) \rangle \\
\langle N \oplus C \Rightarrow & \text{ rec } X. (N \Rightarrow \text{ none } \oplus C \Rightarrow \text{ none } \oplus E \Rightarrow E ; X) \parallel \text{ none} \rangle \\
\langle \text{ none } \Rightarrow & \text{ none } \parallel \text{ none} \rangle
\end{align*}
\]
State: \( N \oplus C \)

**Configurations:**

\[
\langle E \Rightarrow \text{rec } X.( N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow \text{E}; X ) || E \Rightarrow ( N \oplus C ) \rangle \\
\langle N \oplus C \Rightarrow \text{rec } X.( N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow \text{E}; X ) || \text{none} \rangle \\
\langle \text{none} \Rightarrow \text{none} || \text{none} \rangle
\]
Configurations:

\[ \langle E \Rightarrow \text{rec } X.( N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle \]

\[ \langle N \oplus C \Rightarrow \text{rec } X.( N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X ) \parallel \text{none} \rangle \]

\[ \langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle \]
State: none

Configurations:

\[ \langle E \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel E \Rightarrow (N \oplus C) \rangle \]
\[ \langle N \oplus C \Rightarrow \text{rec } X. (N \Rightarrow \text{none} \oplus C \Rightarrow \text{none} \oplus E \Rightarrow E; X) \parallel \text{none} \rangle \]
\[ \langle \text{none} \Rightarrow \text{none} \parallel \text{none} \rangle \]
State: none

Configurations:

\[
\begin{align*}
&E \Rightarrow \text{rec } X.(\ N \Rightarrow \text{none } \oplus C \Rightarrow \text{none } \oplus E \Rightarrow E ; X) \ || \ E \Rightarrow (N \oplus C) \\
&\langle N \oplus C \Rightarrow \text{rec } X.(\ N \Rightarrow \text{none } \oplus C \Rightarrow \text{none } \oplus E \Rightarrow E ; X) \ || \ \text{none} \rangle \\
&\langle \text{none } \Rightarrow \text{none } \ || \ \text{none} \rangle
\end{align*}
\]
Related Work

Krishnaswami, Turon, Dreyer, Garg. **Superficially Substructural Types.** ICFP 2012.


- Powerful generalization of split and merge operations (using commutative monoids) that enables expressive and precise descriptions of sharing.


- References extended with predicate (for expressing local knowledge), rely and guarantee relations to handle sharing of state.

( Paper includes additional Related Work. )
Related Work

Krishnaswami, Turon, Dreyer, Garg. **Superficially Substructural Types.** ICFP 2012.


- Powerful generalization of split and merge operations (using commutative monoids) that enables expressive and precise descriptions of sharing.


- References extended with predicate (for expressing local knowledge), rely and guarantee relations to handle sharing of state.

- We are limited to finite state representations, i.e. typestates.

+ Protocols can express changes over time (“temporal sharing”), without requiring the use of auxiliary variables to distinguish steps.

+ Sharing is a typing artifact and is not tied to a module.

+ Can be type checked without manual intervention.
Summary

- **Contribution:** novel interference-control mechanism, *Rely-Guarantee Protocols*, to control sharing of state mutable by statically disconnected variables.

- **Topics Covered:** (more details in the paper)
  1. Protocol Specification ("public changes")
  2. Protocol Use ("private changes")
  3. Protocol Conformance ("alias interleaving")

Experimental Prototype Implementation:

http://deaf-parrot.googlecode.com