An operational semantics for the C99 restrict type qualifier

Ties Klappe

Radboud University

May 7$^{\text{th}}$ 2024

A motivating example

- **Aliasing**: different symbolic names refer to the same object
- **Pointee**: the object pointed to by a pointer

```c
int foo1(int* p, int* q) {
    *p = 10;
    *q = 11;
    return *p; // if p and q do not alias, *p must evaluate to 10
               // if p and q alias, *p must evaluate to 11
}
```

A motivating example

- **Restrict**: programmer-provided information to inform the compiler specific pointers do not alias under certain conditions

```
// Programmer: hi compiler! I promise you p and q will not alias
int foo2(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p; // if p and q do not alias, *p must evaluate to 10
               // if p and q alias, *p must evaluate to 11
}
```

p $\longrightarrow$ $\boxed{10}$ q $\longrightarrow$ $\boxed{11}$

A motivating example

```
// Programmer: hi compiler! I promise you p and q will not alias
// Compiler: nice! Thanks to this information, I optimized your code
int foo2_optimized(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p 10;
}
```

$p \longrightarrow \boxed{10} \qquad q \longrightarrow \boxed{11}$

The promise can be broken

```c
int foo1(int* p, int* q) {
    *p = 10;
    *q = 11;
    return *p;
}
```

```c
int foo2(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p 10;
}
```

The promise can be broken

```c
int foo1(int* p, int* q) {
    *p = 10;
    *q = 11;
    return *p;
}
```

```c
int foo2(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p 10;
}
```

```c
int main() {
    int x;
    printf("%d, %d\n", foo1(&x, &x), foo2(&x, &x));
}
```

p,q $\rightarrow \boxed{11}$

- Prints $11, 10$, *i.e.* the optimized code has a different result than the original code
- Is the optimization incorrect?

Undefined behavior

- The programmer **broke** the promise by making p and q alias
- This induces **undefined behavior (UB, ☣)**
  - The compiler may *assume* a program is free of UB
  - It does not need to consider such programs when justifying optimizations (*i.e.* the introductory optimization is sound)
- In this presentation we only consider UB induced by restrict, but many other kinds exist (uninitialized memory loads, signed integer overflow, out-of-bounds accesses, ...)

Undefined behavior

- To understand what uses of restrict induce undefined behavior, one should consult the ISO standard

6.7.3.1 Formal definition of `restrict`

1 Let **`D`** be a declaration of an ordinary identifier that provides a means of designating an object **`P`** as a restrict-qualified pointer to type **`T`**.

⋮

3 In what follows, a pointer expression **`E`** is said to be *based* on object **`P`** if (at some sequence point in the execution of **`B`** prior to the evaluation of **`E`**) modifying **`P`** to point to a copy of the array object into which it formerly pointed would change the value of **`E`**.[137] Note that "based" is defined only for expressions with pointer types.

4 During each execution of **`B`**, let **`L`** be any lvalue that has **`&L`** based on **`P`**. If **`L`** is used to access the value of the object **`X`** that it designates, and **`X`** is also modified (by any means), then the following requirements apply: **`T`** shall not be const-qualified. Every other lvalue used to access the value of **`X`** shall also have its address based on **`P`**. Every access that modifies **`X`** shall be considered also to modify **`P`**, for the purposes of this subclause. If **`P`** is assigned the value of a pointer expression **`E`** that is based on another restricted pointer

⋮

6.7.3.1 Formal definition of `restrict`

- Four N-documents submitted since 2018
- Gustedt (2024)[1]: "By its title it is a promise (to provide a formal definition) but it is in fact very delicate mix up of semantic concepts that make it almost impossible to comprehend from the given text."
- MacDonald *et al.* (2022, 2024)[2] report a bug in the definition of "based on"

1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.

⋮

3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.[137)] Note that "based" is defined only for expressions with pointer types.

4 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer

⋮

[1] Gustedt, The semantics of the restrict qualifier

[2] MacDonald, Tong, and Uecker, Defect With Wording Of restrict Specification

Goals

We want a definition for restrict which is:

1. **Unambiguous**, *i.e.* a formal semantics
2. **Consistent** with the standard definition (to the extent possible) and/or existing compiler optimizations
3. **Executable** such that one can test a program for UB
4. **Suitable** to be used for proving compiler optimizations correct (future work)

Approach (formal semantics)

- A vast landscape of formal semantics exists for C, *e.g.* CompCert, CH_2O and Cerberus
- Most of these projects have omitted restrict, except the executable C-in-$\mathbb{K}$ semantics

[1]Hathhorn, Ellison, and Roșu, "Defining the undefinedness of C".

Approach (formal semantics)

- A vast landscape of formal semantics exists for C, *e.g.* CompCert, CH_2O and Cerberus
- Most of these projects have omitted restrict, except the executable C-in-$\mathbb{K}$ semantics
- The paper[1] contains only a single paragraph on restrict, an extensive evaluation reveals several problems (2)
- As a rewrite-based semantics, it is not suitable for reasoning about optimization correctness à la CompCert (4)
  1. Unambiguous: ✓
  2. Consistent: ✗
  3. Executable: ✓
  4. Suitable: !

[1] Hathhorn, Ellison, and Roșu, "Defining the undefinedness of C".

Contributions

- Extensive evaluation
- Identify six problems
- Solve the problems (**consistency**, goal 2)

C-in-$\mathbb{K}$ restrict fragment → Functional C-in-$\mathbb{K}$ restrict fragment → Crestrict

- Understand the semantics
- Redevelop the semantics closer to CompCert style (**suitability**, goal 4)

- Integrate in a big-step semantics
- Interpreter implementation (**executable**, goal 3)

Restrict definition (simplified)

- A pointer is "based on" a restrict pointer if it depends on its value:
  ```
  int x; int* restrict p = &x; int* q = p; // q is based on p
  ```
- A **promise** that a restrict qualified pointer and pointers "based on" it will **not alias** with other pointers during the **scope** it is alive if:
  - The pointer is used to **access** the object it points to
  - The object pointed to is **modified** (by any means)

The language[1] (simplified)

Types

$$\begin{array}{lll} st \in \mathit{SimpleType} & ::= & \mathsf{I32} \mid \mathsf{Ptr}\ \tau \\ \tau_q \in \mathit{TypeQualifier} & ::= & \mathsf{NoRestrict} \mid \mathsf{Restrict} \\ \tau \in \mathit{Type} & ::= & (st, \tau_q) \end{array}$$

[1] A small language based on Blazy and Leroy, "Mechanized semantics for the Clight subset of the C language"

The C-in-$\mathbb{K}$ semantics

Two features are jointly used to support restrict:

1. Pointer values have some extra information, called **bases**
   - Tracks on which restrict qualified pointer(s) a pointer is based
   - Used to distinguish pointers to the same address

$$
\begin{array}{lll}
b \in \mathit{Block}, si \in \mathit{ScopeId} & := & \mathbb{Z} \\
bas \in \mathit{Bases} & := & \mathit{Set}(\mathit{Block} \times \mathit{ScopeId}) \\
 & & \\
\mathit{Val} & ::= & \mathrm{Ptr}\ (\mathit{Block} \times \mathit{Bases}) \mid \ldots
\end{array}
$$

The C-in-$\mathbb{K}$ semantics

Two features are jointly used to support restrict:

1. Pointer values have some extra information, called **bases**
   - Tracks on which restrict qualified pointer(s) a pointer is based
   - Used to distinguish pointers to the same address

$$
\begin{array}{lll}
b \in \mathit{Block}, si \in \mathit{ScopeId} & := & \mathbb{Z} \\
bas \in \mathit{Bases} & := & \mathit{Set}(\underbrace{\mathit{Block}}_{\text{Address of restrict pointer}} \times \underbrace{\mathit{ScopeId}}_{\text{Restrict pointer declaration scope}}) \\
\mathit{Val} & ::= & \mathrm{Ptr}\ (\underbrace{\mathit{Block}}_{\text{Address of the pointee}} \times \mathit{Bases}) \mid \ldots
\end{array}
$$

The C-in-$\mathbb{K}$ semantics

Two features are jointly used to support restrict:

2. The **restrict stack** tracks what memory accesses are allowed by maintaining a per-location **restrict state**

$$\begin{aligned} \mathit{RestrictState} \quad & ::= \quad \text{OnlyRead } \mathit{bas} \mid \text{Restricted } \mathit{bas} \mid \text{Unrestricted} \\ R \in \mathit{RestrictStack} \quad & := \quad \mathit{List}(\mathit{ScopeId} \times (\mathit{Block} \to \mathit{RestrictState})) \end{aligned}$$

The C-in-$\mathbb{K}$ semantics

Two features are jointly used to support restrict:

2. The **restrict stack** tracks what memory accesses are allowed by maintaining a per-location **restrict state**

$$\begin{aligned}
\mathit{RestrictState} &::= \underbrace{\text{OnlyRead } \mathit{bas}}_{\text{Load via Ptr } (_, \mathit{bas})} \mid \underbrace{\text{Restricted } \mathit{bas}}_{\text{Store via Ptr } (_, \mathit{bas})} \mid \underbrace{\text{Unrestricted}}_{\text{Loads via pointers with different bases}} \\
R \in \mathit{RestrictStack} &:= \mathit{List}(\underbrace{\mathit{ScopeId}}_{\text{Scope in which the access occured}} \times (\mathit{Block} \to \mathit{RestrictState}))
\end{aligned}$$

The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}

// Scope si_main
int main() {
    int x;
    foo(&x, &x);
}
```

M:

$\emptyset$

R:

$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}

// Scope si_main
int main() {
    int x; // &x = b_x
    foo(&x, &x);
}
```

M:

$\{b_x \mapsto \text{Undef}\}$

R:

$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo      &p = b_p      &q = b_q
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}

// Scope si_main
int main() {
    int x; // &x = b_x
    foo(&x, &x);
}
```

M:

$$\{b_p \mapsto \mathrm{Ptr}\ (b_x, \{(b_p, \mathtt{si}_{\mathtt{foo}})\}),\ b_q \mapsto \mathrm{Ptr}\ (b_x, \{(b_q, \mathtt{si}_{\mathtt{foo}})\}),\ b_x \mapsto \mathrm{Undef}\}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\emptyset$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo      &p = b_p      &q = b_q
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}
```

```
// Scope si_main
int main() {
    int x; // &x = b_x
    foo(&x, &x);
}
```

M:

$$\{b_p \mapsto \mathrm{Ptr}\ (b_x, \{(b_p, \mathtt{si}_{\mathtt{foo}})\}),\ b_q \mapsto \mathrm{Ptr}\ (b_x, \{(b_q, \mathtt{si}_{\mathtt{foo}})\}),\ b_x \mapsto 10\}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\{b_x \mapsto \mathrm{Restricted}\ \{(b_p, \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo      &p = b_p      &q = b_q
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}

// Scope si_main
int main() {
    int x; // &x = b_x
    foo(&x, &x);
}
```

$\text{Restricted } \{(b_q, \mathtt{si}_{\mathtt{foo}})\} \sqcup \text{Restricted } \{(b_p, \mathtt{si}_{\mathtt{foo}})\} = \ldots$

M:

$$\begin{array}{l} \{b_p \mapsto \text{Ptr } (b_x, \{(b_p, \mathtt{si}_{\mathtt{foo}})\}), \\ \ b_q \mapsto \text{Ptr } (b_x, \{(b_q, \mathtt{si}_{\mathtt{foo}})\}), \\ \ b_x \mapsto \text{Undef}\} \end{array}$$

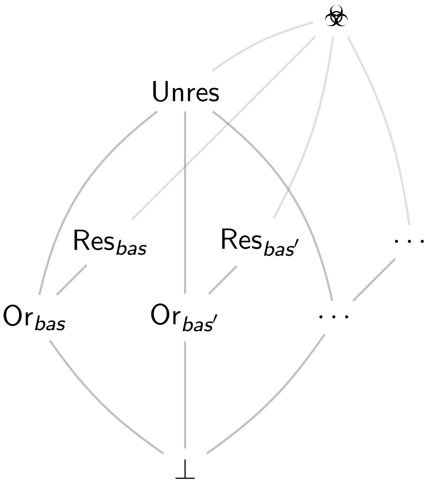
R:

$\mathtt{si}_{\mathtt{foo}}$	$\{b_x \mapsto \text{Restricted } \{(b_p, \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

The introductory example under the C-in-$\mathbb{K}$ semantics

$\text{Restricted } \{(b_q, \texttt{si}_{\texttt{foo}})\} \sqcup \text{Restricted } \{(b_p, \texttt{si}_{\texttt{foo}})\} = \ldots$

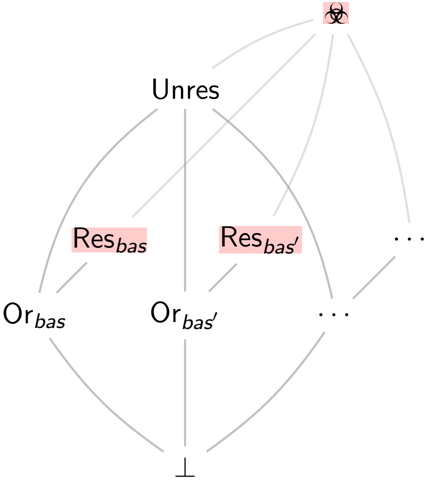
- The symmetric $\sqcup$ operation describes the result of joining two restrict states
- Unres = Unrestricted, Res_{bas} = (Restricted *bas*) and Or_{bas} = (OnlyRead *bas*)
- $bas \neq bas'$



The introductory example under the C-in-$\mathbb{K}$ semantics

$\text{Restricted}\ \{(b_q, \texttt{si}_{\texttt{foo}})\} \sqcup \text{Restricted}\ \{(b_p, \texttt{si}_{\texttt{foo}})\} = ☣$

- The symmetric $\sqcup$ operation describes the result of joining two restrict states
- Unres = Unrestricted, Res_{bas} = (Restricted *bas*) and Or_{bas} = (OnlyRead *bas*)
- $bas \neq bas'$



The introductory example under the C-in-$\mathbb{K}$ semantics

```
// Scope si_foo      &p = b_p      &q = b_q
int foo(int* restrict p, int* restrict q) {
    *p = 10;
    *q = 11;
    return *p;
}

// Scope si_main
int main() {
    int x; // &x = b_x
    foo(&x, &x);
}
```

UB: Restricted $\{(b_q, \texttt{si}_{\texttt{foo}})\} \sqcup$ Restricted $\{(b_p, \texttt{si}_{\texttt{foo}})\} = ☣$

M:

$$\boxed{\begin{array}{l}\{b_p \mapsto \mathrm{Ptr}\ (b_x, \{(b_p, \texttt{si}_{\texttt{foo}})\}), \\ \ b_q \mapsto \mathrm{Ptr}\ (b_x, \{(b_q, \texttt{si}_{\texttt{foo}})\}), \\ \ b_x \mapsto \mathrm{Undef}\}\end{array}}$$

R:

$\texttt{si}_{\texttt{foo}}$	$\{b_x \mapsto$ Restricted $\{(b_p, \texttt{si}_{\texttt{foo}})\}\}$
$\texttt{si}_{\texttt{main}}$	$\emptyset$

Evaluating the C-in-$\mathbb{K}$ semantics

- The semantics correctly gives undefined behavior to our introductory example!
  But, we argue, there are some problems:
- Too much undefined behavior (TMU)
  - Aliasing loads
  - Returning restrict pointers
- Too little undefined behavior (TLU)
  - Array of restrict pointers
  - Nested restrict pointers
  - Semantic preservation under inlining
  - Call to free

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}
// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

$\texttt{q} \longrightarrow x \qquad \texttt{r,s} \rightarrow y$

- Simplified version of example 3 from the ISO standard demonstrating DB
- y does **not** get **modified** in the scope $\texttt{si}_{\texttt{h}}$ of `r,s`

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$\boxed{\emptyset}$

R:

$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$\{b_x \mapsto \text{Undef}, b_y \mapsto \text{Undef}\}$

R:

$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}
```

```
// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{b_x \mapsto \text{Undef}, \\ b_y \mapsto \text{Undef}, \\ b_p \mapsto \text{Ptr}\ (b_y, \{(b_p, \mathtt{si}_{\mathtt{main}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{b_x \mapsto \text{Undef}, \; b_y \mapsto 0, \; b_p \mapsto \text{Ptr}\,(b_y, \{(b_p, \mathtt{si}_{\mathtt{main}})\})\}$$

R:

$\mathtt{si}_{\mathtt{main}}$ $\{b_y \mapsto \text{Restricted}\,\{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\begin{array}{l} \{b_x \mapsto \text{Undef},\ b_y \mapsto 0, \\ \ b_p \mapsto \text{Ptr}\ (b_y, \{(b_p, \mathtt{si}_{\mathtt{main}})\}), \\ \ b_q \mapsto \text{Ptr}\ (b_x, \emptyset), \\ \ b_r \mapsto \text{Ptr}\ (b_y, \{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}), \\ \ b_s \mapsto \text{Ptr}\ (b_y, \{(b_s, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}) \\ \} \end{array}$$

R:

$\mathtt{si}_{\mathtt{h}}$	$\emptyset$
$\mathtt{si}_{\mathtt{main}}$	$\{b_y \mapsto \text{Restricted}\ \{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{b_x \mapsto \mathsf{Undef},\ b_y \mapsto 0,\ b_p \mapsto \mathsf{Ptr}\ (b_y, \{(b_p, \mathtt{si}_{\mathtt{main}})\}),\ b_q \mapsto \mathsf{Ptr}\ (b_x, \emptyset),\ b_r \mapsto \mathsf{Ptr}\ (b_y, \{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}),\ b_s \mapsto \mathsf{Ptr}\ (b_y, \{(b_s, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\})\}$$

R:

$\mathtt{si}_{\mathtt{h}}$	$\{b_y \mapsto \mathsf{OnlyRead}\ \{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\{b_y \mapsto \mathsf{Restricted}\ \{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}
```

$\text{OnlyRead } \{(b_r, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\} \sqcup$
$\text{OnlyRead } \{(b_s, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\} = \ldots$

M:

$$\{\ldots,\; b_r \mapsto \text{Ptr } (b_y, \{(b_r, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\}),\; b_s \mapsto \text{Ptr } (b_y, \{(b_s, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\})\ \}$$

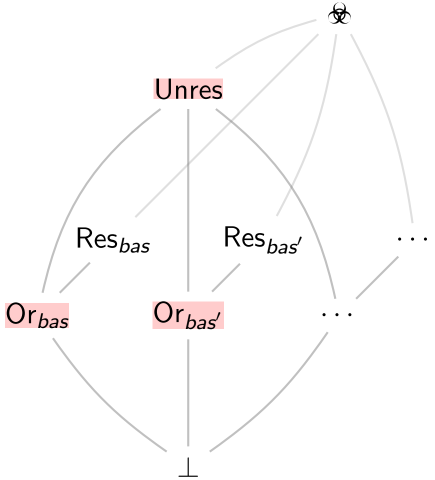
```
// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

R:

$\texttt{si}_\texttt{h}$	$\{b_y \mapsto \text{OnlyRead } \{(b_r, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\}\}$
$\texttt{si}_\texttt{main}$	$\{b_y \mapsto \text{Restricted } \{(b_p, \texttt{si}_\texttt{main})\}\}$

Aliasing loads (TMU)

$$\mathsf{OnlyRead}\ \{(b_r, \mathtt{si_h}), (b_p, \mathtt{si_{main}})\} \sqcup\ \mathsf{OnlyRead}\ \{(b_s, \mathtt{si_h}), (b_p, \mathtt{si_{main}})\} = \mathsf{Unrestricted}$$



Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{\ldots, \; b_r \mapsto \text{Ptr}\ (b_y, \{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}), \; b_s \mapsto \text{Ptr}\ (b_y, \{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\})\ \}$$

R:

$\mathtt{si}_\mathtt{h}$	$\{b_y \mapsto \text{Unrestricted}\}$
$\mathtt{si}_\mathtt{main}$	$\{b_y \mapsto \text{Restricted}\ \{(b_p, \mathtt{si}_\mathtt{main})\}\}$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}
```

```
// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{\ldots,\ b_r \mapsto \mathrm{Ptr}\ (b_y, \{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}),\ b_s \mapsto \mathrm{Ptr}\ (b_y, \{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\})\ \}$$

R:

$\mathtt{si}_\mathtt{h}$	$\{\ldots,\ b_y \mapsto \mathrm{Unrestricted}\}$
$\mathtt{si}_\mathtt{main}$	$\{b_y \mapsto \mathrm{Restricted}\ \{(b_p, \mathtt{si}_\mathtt{main})\}\}$

Aliasing loads (TMU)

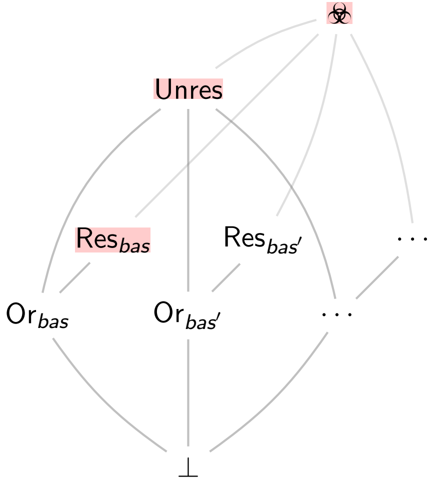
- $\mathtt{si}_{\mathtt{h}}$ is part of the execution of scope $\mathtt{si}_{\mathtt{main}}$
- Join the restrict states when $\mathtt{si}_{\mathtt{h}}$ terminates!

$\mathtt{si}_{\mathtt{h}}$	$\{b_y \mapsto \text{Unrestricted}\}$
$\mathtt{si}_{\mathtt{main}}$	$\{b_y \mapsto \text{Restricted } \{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

$\sqcup$

Aliasing loads (TMU)

$$\text{Restricted } \{(b_p, \texttt{si}_{\texttt{main}})\} \sqcup \text{Unrestricted} = ☣$$



Aliasing loads (TMU)

- The fundamental problem with Unrestricted is **information loss**
- Idea: **remove** Unrestricted entirely and promote (OnlyRead *bas*) to (OnlyRead *fbas*) with $fbas \in Set(Bases)$, *i.e.* a **family of sets of bases**
  - Every set of the family represents a pointer used for a load
  - if $|F_{bas}| > 1$ the semantics of Unrestricted apply
  - if $F_{bas} = \{bas\}$ the semantics of (OnlyRead *bas*) apply

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}
```

$\mathsf{OnlyRead}\ \{\{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}\} \sqcup$
$\mathsf{OnlyRead}\ \{\{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}\} = \ldots$

M:

$$\{\ldots, \\ b_r \mapsto \mathsf{Ptr}\ (b_y, \{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}), \\ b_s \mapsto \mathsf{Ptr}\ (b_y, \{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\})\ \}$$

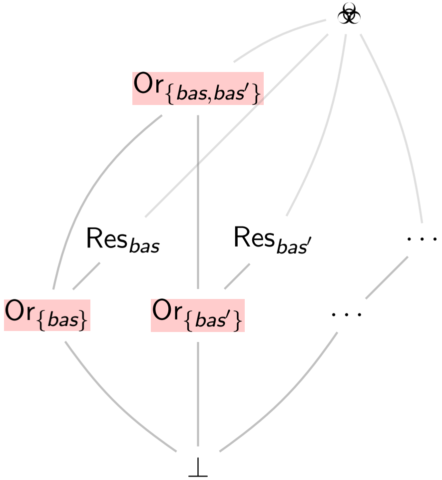
R:

$\mathtt{si}_\mathtt{h}$	$\{b_y \mapsto \mathsf{OnlyRead}\ \{\{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}\}\}$
$\mathtt{si}_\mathtt{main}$	$\{b_y \mapsto \mathsf{Restricted}\ \{(b_p, \mathtt{si}_\mathtt{main})\}\}$

```
// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

Aliasing loads (TMU)

$$\mathsf{OnlyRead}\ \{\{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}\} \sqcup \mathsf{OnlyRead}\ \{\{(b_s, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}\} = \ldots$$



- The updated symmetric $\sqcup$ operation (simplified)
- $bas \neq bas'$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{\ldots,\ b_r \mapsto \text{Ptr}\ (b_y, \{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}),\ b_s \mapsto \text{Ptr}\ (b_y, \{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\})\ \}$$

R:

$\mathtt{si}_\mathtt{h}$	$\{b_y \mapsto \text{OnlyRead}\ \{\ \{(b_r, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\},\ \{(b_s, \mathtt{si}_\mathtt{h}), (b_p, \mathtt{si}_\mathtt{main})\}\ \}\ \}$
$\mathtt{si}_\mathtt{main}$	$\{b_y \mapsto \text{Restricted}\ \{(b_p, \mathtt{si}_\mathtt{main})\}\}$

Aliasing loads (TMU)

```
// Scope si_h
void h(int* q, int* restrict r, int* restrict s) {
    *q = *r + *s;
}

// Scope si_main
int main() {
    int x, y;
    int* restrict p = &y;
    *p = 0;
    h(&x, p, p);
}
```

M:

$$\{\ldots,\ b_r \mapsto \text{Ptr}\ (b_y, \{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}),\ b_s \mapsto \text{Ptr}\ (b_y, \{(b_s, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{h}}$	$\{\ldots,\ b_y \mapsto \text{OnlyRead}\ \{ \{(b_r, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}, \{(b_s, \mathtt{si}_{\mathtt{h}}), (b_p, \mathtt{si}_{\mathtt{main}})\}\ \}\ \}$
$\mathtt{si}_{\mathtt{main}}$	$\{b_y \mapsto \text{Restricted}\ \{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

Aliasing loads (TMU)

- Recall that the restrict rules only apply during the **scope a restrict pointer is alive**
- Filtering: when joining between scopes, remove bases from the expired scope $\mathtt{si}_{\mathtt{h}}$

Aliasing loads (TMU)

- Recall that the restrict rules only apply during the **scope a restrict pointer is alive**
- Filtering: when joining between scopes, remove bases from the expired scope $\texttt{si}_\texttt{h}$

$\texttt{si}_\texttt{h}$
$$\{b_y \mapsto \text{OnlyRead } \{ \{(b_r, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\}, \{(b_s, \texttt{si}_\texttt{h}), (b_p, \texttt{si}_\texttt{main})\} \} \}$$

$\texttt{si}_\texttt{main}$
$$\{b_y \mapsto \text{Restricted } \{(b_p, \texttt{si}_\texttt{main})\}\}$$

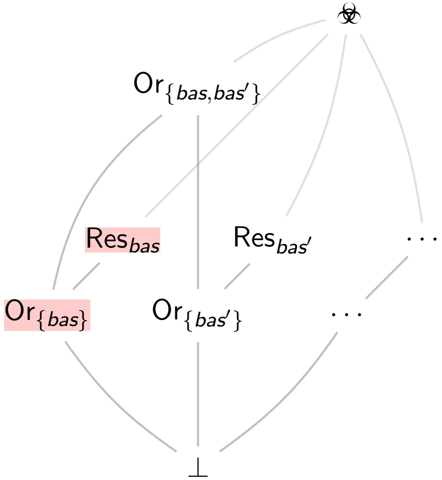
Filter expired bases + $\sqcup$

- Filtered state: OnlyRead $\{\{(b_p, \texttt{si}_\texttt{main})\}\}$
- OnlyRead $\{\{(b_p, \texttt{si}_\texttt{main})\}\} \sqcup$ Restricted $\{(b_p, \texttt{si}_\texttt{main})\} = \ldots$

Aliasing loads (TMU)

$$\text{OnlyRead } \{\{(b_p, \texttt{si}_{\texttt{main}})\}\} \sqcup \text{Restricted } \{(b_p, \texttt{si}_{\texttt{main}})\} = \text{Restricted } \{(b_p, \texttt{si}_{\texttt{main}})\}$$

- The updated symmetric $\sqcup$ operation (simplified)
- $bas \neq bas'$



Aliasing loads (TMU)

- Loads via aliased pointers are now permitted ☺
- Achieved our goal of relaxing the semantics for this problem to give **less UB**, **consistent** with the ISO standard

Crestrict refinements

- Too much undefined behavior (TMU)
  - Aliasing loads: **adjust restrict states and $\sqcup$ lattice**
  - Returning restrict pointers: **track active scopes and filter pointer values**
- Too little undefined behavior (TLU)
  - Array of restrict pointers: **refine bases granularity to offsets**
  - Nested restrict pointers: **missing subclause and pointer values as a tree structure**
  - Semantic preservation under inlining: **deferred $\rightarrow$ eager check**
  - Call to free: **update the restrict state**
- **Consistency**, goal 2 ✓(*i.e.*, to the best of our knowledge)

Evaluation

- Implemented the semantics in an interpreter, written in Rust (**executable**, goal 3 ✓)
- A (public) test suite dedicated to restrict does not exist
- Created our own suite of 96 tests, build around common restrict use cases and the discussed problems

Conclusion

- Redeveloped the restrict fragment of the C-in-$\mathbb{K}$ semantics in a functional style (4)
- We argued it has six consistency problems
- We proposed changes to the semantic domains and rules to solve them (2)
- The new Crestrict semantics (1,3) were implemented in an interpreter and evaluated under a more extensive test suite

1. Unambiguous: ✓
2. Consistent: ✓
3. Executable: ✓
4. Suitable: ✓

Future work

- Assignments between restrict pointers
- A more complete language
- Proving optimizations correct (the sequal of goal 4)
- ...

Returning restrict pointers (TMU)

```
int* as_mut_ptr(int* restrict v) {
    return v;
}
```

$\text{Restricted } \{(b_{v2}, \mathtt{si}_{\mathtt{as\_mut\_ptr\_2}})\} \sqcup$
$\text{Restricted } \{(b_{v1}, \mathtt{si}_{\mathtt{as\_mut\_ptr\_1}})\} = ☣$

```
int main() {
    int a;

    int* p = as_mut_ptr(&a);
    int* q = as_mut_ptr(&a);

    *p = 0;
    *q = 0;
}
```

M:

$$\boxed{\begin{aligned}\{&b_a \mapsto 0,\\ &b_p \mapsto \text{Ptr } (b_a, \{(b_{v1}, \mathtt{si}_{\mathtt{as\_mut\_ptr\_1}})\}),\\ &b_q \mapsto \text{Ptr } (b_a, \{(b_{v2}, \mathtt{si}_{\mathtt{as\_mut\_ptr\_2}})\})\ \}\end{aligned}}$$

R:

$\mathtt{si}_{\mathtt{main}}$ $\boxed{\{b_a \mapsto \text{Restricted } \{(b_{v1}, \mathtt{si}_{\mathtt{as\_mut\_ptr\_1}})\}\}}$

Array of restrict pointers (TLU)

```
// Scope si_main
int main() {
    int x;
    int* restrict a[2] = {&x, &x};

    *(a[0]) = 10;
    *(a[1]) = 11;
}
```

$\text{Restricted } \{(b_a, \mathtt{si}_{\mathtt{main}})\} \sqcup \text{Restricted } \{(b_a, \mathtt{si}_{\mathtt{main}})\}$
$= \text{Restricted } \{(b_a, \mathtt{si}_{\mathtt{main}})\}$

M:

$$\boxed{\begin{aligned} \{ & b_x \mapsto 10, \\ & b_a \mapsto \{\text{Ptr }(b_x, \{(b_a, \mathtt{si}_{\mathtt{main}})\}), \\ & \qquad\quad \text{Ptr }(b_x, \{(b_a, \mathtt{si}_{\mathtt{main}})\})\ \}\} \end{aligned}}$$

R:

$\mathtt{si}_{\mathtt{main}}$	$\{b_x \mapsto \text{Restricted } \{(b_a, \mathtt{si}_{\mathtt{main}})\}\}$

Semantic preservation under inlining (TLU)

```c
// Scope si_foo
void foo(int* q) {
    *q = 0;
    while(1) {}
    // Never terminates
}

// Scope si_main
int main() {
    int x = 5;
    int* restrict p = &x;
    *p;
    foo(&x);
}
```

M:

$$\{b_x \mapsto 5,\ b_p \mapsto \mathrm{Ptr}\ (b_x, \{(b_p, \mathtt{si_{main}})\}),\ b_q \mapsto \mathrm{Ptr}\ (b_x, \emptyset)\ \}$$

R:

$\mathtt{si_h}$	$\{b_x \mapsto \mathrm{Restricted}\ \emptyset\}$
$\mathtt{si_{main}}$	$\{b_x \mapsto \mathrm{OnlyRead}\ \{(b_p, \mathtt{si_{main}})\}\}$

Semantic preservation under inlining (TLU)

```
// foo is inlined into main
// Scope si_main
int main() {
    int x = 5;
    int* restrict p = &x;
    *p;
    int* q = &x;
    *q = 0;
    while (1) {}
}
```

$\text{Restricted } \emptyset \sqcup \text{OnlyRead } \{(b_p, \mathtt{si}_{\mathtt{main}})\} = ☣$

M:

$$\boxed{\begin{array}{l}\{b_x \mapsto 5,\\ \ b_p \mapsto \text{Ptr } (b_x, \{(b_p, \mathtt{si}_{\mathtt{main}})\}),\\ \ b_q \mapsto \text{Ptr } (b_x, \emptyset)\ \}\end{array}}$$

R:

$\mathtt{si}_{\mathtt{main}}$	$\{b_x \mapsto \text{OnlyRead } \{(b_p, \mathtt{si}_{\mathtt{main}})\}\}$

Call to free (TLU)

```
// Scope si_bar
void bar(int* s) {
    free(s);
}
// Scope si_foo
void foo(int* restrict q, int* r) {
    *q = 5;
    bar(r);
}
// Scope si_main
int main() {
    // Stored at b_v
    int* p = malloc(sizeof(int));
    foo(p, p);
}
```

M:

$$\{b_v \mapsto 5,\ b_p \mapsto \mathsf{Ptr}\ (b_v, \emptyset),\ b_q \mapsto \mathsf{Ptr}\ (b_v, \{(b_q, \mathtt{si}_{\mathtt{foo}})\}),\ b_r \mapsto \mathsf{Ptr}\ (b_v, \emptyset)\ \}$$

R:

$\mathtt{si}_{\mathtt{h}}$	$\{b_v \mapsto \mathsf{Restricted}\ (b_q, \mathtt{si}_{\mathtt{foo}})\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

```
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p; // Optimized to 10 by GCC
}
// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

$\texttt{p,q} \rightarrow \texttt{xp} \longrightarrow \texttt{x}$

- UB due to a subtle subclause of the standard

Restrict definition (simplified)

- A pointer is "based on" a restrict pointer if it depends on its value:

```
int x; int* restrict p = &x; int* q = p; // q is based on p
```

- A **promise** from the programmer to the compiler that a restrict qualified pointer and pointers "based on" it will **not alias** with other pointers during the **scope** it is alive if:
  - The pointer is used to **access** the object it points to
  - The object pointed to is **modified** (by any means)
- "Modifications of the object pointed to by a restrict pointer are considered to modify the restrict pointer object itself"

Nested restrict pointers (TLU)

- What does "modifications of the object pointed to by a restrict pointer are considered to modify the restrict pointer object itself" mean?
- Modifications are represented by the restrict state Restricted

Nested restrict pointers (TLU)

- What does "modifications of the object pointed to by a restrict pointer are considered to modify the restrict pointer object itself" mean?
- Modifications are represented by the restrict state Restricted

```
// Scope si_main
{
int x; // &x = b_x
int* restrict p = &x; // &p = b_p
*p = 10; // Modification
}
```

M:

$$\{b_x \mapsto 10, \; b_p \mapsto \text{Ptr } (b_x, \{(b_p, \texttt{si}_{\texttt{main}})\}) \}$$

R:

$\texttt{si}_{\texttt{main}}$ $\{b_x \mapsto \text{Restricted } \{(b_p, \texttt{si}_{\texttt{main}})\} \}$

Nested restrict pointers (TLU)

- What does "modifications of the object pointed to by a restrict pointer are considered to modify the restrict pointer object itself" mean?
- Modifications are represented by the restrict state Restricted

```
// Scope si_main
{
int x; // &x = b_x
int* restrict p = &x; // &p = b_p
*p = 10; // Modification
}
```

M:

$$\{b_x \mapsto 10,\ b_p \mapsto \mathrm{Ptr}\ (b_x, \{(b_p, \mathtt{si}_{\mathtt{main}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{main}}$

$$\{b_x \mapsto \mathrm{Restricted}\ \{(b_p, \mathtt{si}_{\mathtt{main}})\},\ b_p \mapsto \mathrm{Restricted}\ \emptyset\ \}$$

Nested restrict pointers (TLU)

```
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p;
}

// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

M:

$$\{\ b_x \mapsto \text{Undef},\ b_{xp} \mapsto \text{Ptr}\ (b_x, \emptyset),\ b_p \mapsto \text{Ptr}\ (b_{xp}, \{(b_p, \mathtt{si}_{\mathtt{foo}})\}),\ b_q \mapsto \text{Ptr}\ (b_{xp}, \{(b_q, \mathtt{si}_{\mathtt{foo}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\{\ldots, b_{xp} \mapsto \text{OnlyRead}\ \{(b_p, \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

```c
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p;
}

// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

M:

$$\{\ b_x \mapsto \mathsf{Undef},\ b_{xp} \mapsto \mathsf{Ptr}\ (b_x, \emptyset),\ b_p \mapsto \mathsf{Ptr}\ (b_{xp}, \{(b_p, \mathtt{si}_{\mathtt{foo}})\}),\ b_q \mapsto \mathsf{Ptr}\ (b_{xp}, \{(b_q, \mathtt{si}_{\mathtt{foo}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\{\ldots, b_{xp} \mapsto \mathsf{OnlyRead}\ \{(b_p, \mathtt{si}_{\mathtt{foo}})\}, b_x \mapsto \mathsf{Restricted}\ \{(b_{xp}, \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

- We now need to change the restrict state of b_{xp} to Restricted
- The only Restricted state joinable with the current state is Restricted $\{(b_p, \mathtt{si}_{\mathtt{foo}})\}$
- Problem: **not enough information** to produce this state, *i.e.* the semantics did a store through Ptr $(b_x, \{(b_{xp}, \mathtt{si}_{\mathtt{foo}})\})$

$\mathtt{si}_{\mathtt{foo}}$	$\{\ldots,\ b_{xp} \mapsto \text{OnlyRead } \{(b_p, \mathtt{si}_{\mathtt{foo}})\},\ b_x \mapsto \text{Restricted } \{(b_{xp}, \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

- Idea: pointer value as a **tree** structure, to track how bases themselves are derived!
- Ptr $(b_x, \{((b_{xp}, \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\}), \mathtt{si}_{\mathtt{foo}})\})$

$$\begin{tikzcd} b_x \arrow[d, dash, dashed] \\ (b_{xp}, \mathtt{si}_{\mathtt{foo}}) \arrow[d, dash, dashed] \\ (b_p, \mathtt{si}_{\mathtt{foo}}) \end{tikzcd}$$

Nested restrict pointers (TLU)

```
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p;
}

// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

M:

$$\{\ b_x \mapsto \mathsf{Undef},\ b_{xp} \mapsto \mathsf{Ptr}\ (b_x, \emptyset),\ b_p \mapsto \mathsf{Ptr}\ (b_{xp}, \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\}),\ b_q \mapsto \mathsf{Ptr}\ (b_{xp}, \{((b_q, \emptyset), \mathtt{si}_{\mathtt{foo}})\})\ \}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\{\ldots,\ b_{xp} \mapsto \mathsf{Restricted}\ \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\},\ b_x \mapsto \mathsf{Restricted}\ \{((b_{xp}, \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\}), \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

```
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p;
}

// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

M:

$$\begin{array}{l} \{\ b_x \mapsto \text{Undef}, \\ \ b_{xp} \mapsto \text{Ptr}\ (b_x, \emptyset), \\ \ b_p \mapsto \text{Ptr}\ (b_{xp}, \{((b_p, \emptyset), \texttt{si}_{\texttt{foo}})\}), \\ \ b_q \mapsto \text{Ptr}\ (b_{xp}, \{((b_q, \emptyset), \texttt{si}_{\texttt{foo}})\})\ \} \end{array}$$

R:

$\texttt{si}_{\texttt{foo}}$	$\{\ldots, b_{xp} \mapsto \text{Restricted}\ \{((b_p, \emptyset), \texttt{si}_{\texttt{foo}})\}$, $b_x \mapsto \text{Restricted}\ \{((b_{xp}, \{((b_p, \emptyset), \texttt{si}_{\texttt{foo}})\}), \texttt{si}_{\texttt{foo}})\}\}$
$\texttt{si}_{\texttt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

```
// Scope si_foo
int foo(int *restrict *restrict p, int *restrict *restrict q) {
    **p = 10;
    **q = 11;
    return **p;
}

// Scope si_main
int main() {
    int x;
    int* xp = &x;
    foo(&xp, &xp);
}
```

$\text{OnlyRead } \{((b_q, \emptyset), \mathtt{si}_{\mathtt{foo}})\} \sqcup \text{Restricted } \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\} = \ldots$

M:

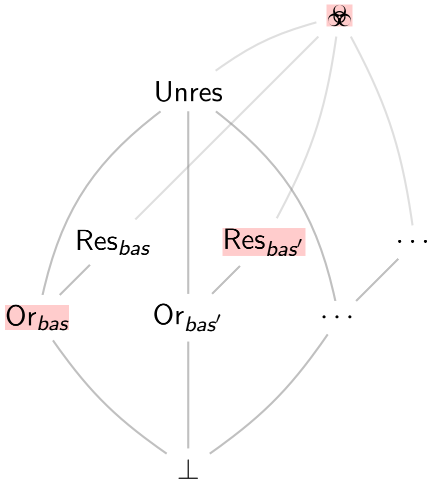
$$\{ \ldots, \; b_q \mapsto \text{Ptr } (b_{xp}, \{((b_q, \emptyset), \mathtt{si}_{\mathtt{foo}})\}) \}$$

R:

$\mathtt{si}_{\mathtt{foo}}$	$\{\ldots, b_{xp} \mapsto \text{Restricted } \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\},$ $b_x \mapsto \text{Restricted } \{((b_{xp}, \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\}), \mathtt{si}_{\mathtt{foo}})\}\}$
$\mathtt{si}_{\mathtt{main}}$	$\emptyset$

Nested restrict pointers (TLU)

$$\mathsf{OnlyRead}\ \{((b_q, \emptyset), \mathtt{si}_{\mathtt{foo}})\} \sqcup \mathsf{Restricted}\ \{((b_p, \emptyset), \mathtt{si}_{\mathtt{foo}})\} = ☣$$



Nested restrict pointers (TLU)

- Implemented a subtle subclause of the standard (in line with the GCC interpretation)
  - Updated pointer values to a **tree-like structure** to track how bases themselves are derived
- Achieved our goal of giving undefined behavior ☺

Where are bases added to the pointer value?

```
// Scope si_main
{
    int x; // &x = b_x
    int* restrict p = &x; // &b_p

    int* q = p; // Propagate the bases to q
    *p = ...;   // Used directly in lvalue position
}
```

$$\dfrac{\dfrac{\dfrac{E(p) = b_p}{G, E \vdash p, \sigma \Downarrow_{\mathsf{L}} (b_p, \emptyset), \sigma'}\ (\mathsf{EId}) \qquad (\mathtt{load}\ \sigma'\ (b_p, \emptyset)) = \mathsf{Ptr}\ (b_x, \emptyset), \sigma'' \qquad \mathtt{is\_restrict}\ (\mathtt{type}\ e)}{G, E \vdash p, \sigma \Downarrow_{\mathsf{R}} \mathtt{add\_prov}\ (\mathsf{Ptr}\ (b_x, \emptyset))\ ((b_p, \emptyset), \mathtt{si}_{\mathtt{main}}), \sigma''}\ (\mathsf{ELvalConvRestrict})}{G, E \vdash *p, \sigma \Downarrow_{\mathsf{L}} (b_x, \{((b_p, \emptyset), \mathtt{si}_{\mathtt{main}})\}), \sigma''}\ (\mathsf{EDeref})$$