Effective Types for C formalized in Coq
(work in progress)

Robbert Krebbers

Radboud University Nijmegen

April 23, 2013 @ TYPES, Toulouse, France

Aliasing

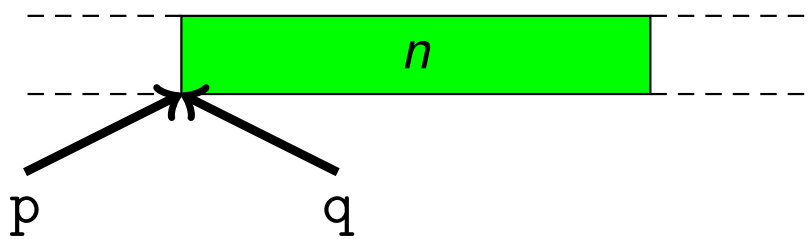
Aliasing: different pointers pointing to the same object

Aliasing

Aliasing: different pointers pointing to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x;
}
```

When called with aliased pointers, the original value *n* of `*p` is returned

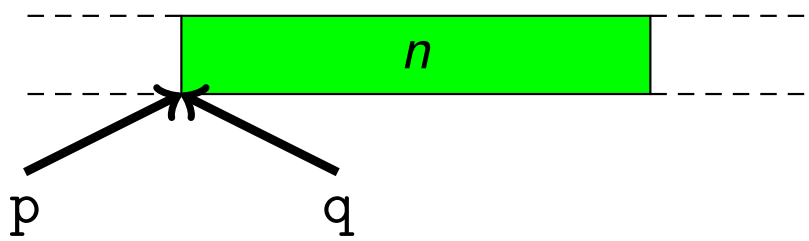


Aliasing

Aliasing: different pointers pointing to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x;
}
```

When called with aliased pointers, the original value n of `*p` is returned



Optimizing `x` away is unsound, as 314 would always be returned

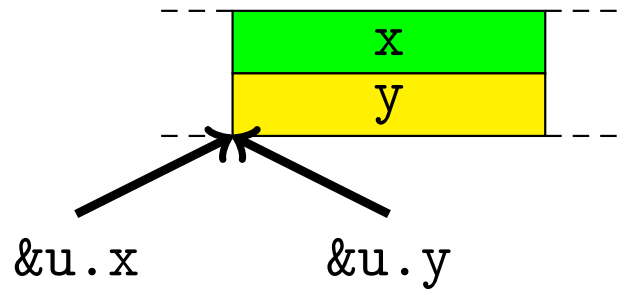
Aliasing with different types

The function

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

can still be called with aliased pointers

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



In C89, this program was "allowed" (and yields 271)

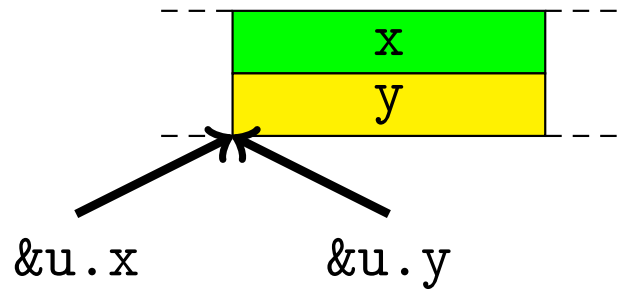
Aliasing with different types

The function

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

can still be called with aliased pointers

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



In C89, this program was "allowed" (and yields 271)

In C11, types can be used for non-aliasing hypotheses:

- Reads/writes "using the wrong type" are "not allowed"
- A compiler can use that to assume that `p` and `q` do not alias

Undefined behavior according to the C standard

- The C standard classifies these "not allowed" programs as *undefined behavior*
- Such programs are not statically excluded
- and may do literally anything when executed

Undefined behavior according to the C standard

- The C standard classifies these "not allowed" programs as *undefined behavior*
- Such programs are not statically excluded
- and may do literally anything when executed
- Compilers are allowed to assume no undefined behavior occurs
- and therefore omit (expensive) dynamic checks

Undefined behavior according to the C standard

- The C standard classifies these "not allowed" programs as *undefined behavior*
- Such programs are not statically excluded
- and may do literally anything when executed
- Compilers are allowed to assume no undefined behavior occurs
- and therefore omit (expensive) dynamic checks

A formal C semantics should thus account for undefined behavior

Other difficulties of the C11 memory model

Interplay between low- and high-level

- Each object *should be* represented as a sequence of bytes
  . . . which can be inspected and manipulated *in C*
- Each object can be accessed using typed expressions
  . . . that are used by compilers for optimizations

Hence, the formal memory model needs to keep track of more information than present in the memory of an actual machine

Other difficulties of the C11 memory model

Interplay between low- and high-level

- Each object *should be* represented as a sequence of bytes
  ... which can be inspected and manipulated *in C*
- Each object can be accessed using typed expressions
  ... that are used by compilers for optimizations

Hence, the formal memory model needs to keep track of more information than present in the memory of an actual machine

The standard is unclear on many of such difficulties. Opportunities for a formal semantics to resolve this unclarity!

Contribution

An abstract formal memory for C supporting

- Types (arrays, structs, unions, . . .)
- Strict aliasing restrictions (effective types)
- Indeterminate values
- Pointers one past
- Byte-level operations
- Parametrized by an interface for machine integers
- Multiple object representations
- Alignment
- Formalized in Coq

How others treat pointers

Many existing formal C semantics (*e.g.* CompCert) treat the memory as containing just bytes:

- The memory is a finite map of cells, each cell consisting of a array of bytes
- Pointers are pairs (x, ofs) where x identifies the cell, and ofs is an offset into that cell

How others treat pointers

Many existing formal C semantics (*e.g.* CompCert) treat the memory as containing just bytes:

- The memory is a finite map of cells, each cell consisting of a array of bytes
- Pointers are pairs (x, ofs) where x identifies the cell, and ofs is an offset into that cell

Too little information to capture strict aliasing restrictions

How we treat pointers

In our approach the memory contains symbolic values:

- The memory is a finite map of cells, each cell consisting of a well-typed tree of bytes
- Pointers are pairs (x, r) where x identifies the cell, and r is a path through the tree

How we treat pointers

In our approach the memory contains symbolic values:

- The memory is a finite map of cells, each cell consisting of a well-typed tree of bytes
- Pointers are pairs (x, r) where x identifies the cell, and r is a path through the tree

Gives a semantics for strict aliasing restrictions

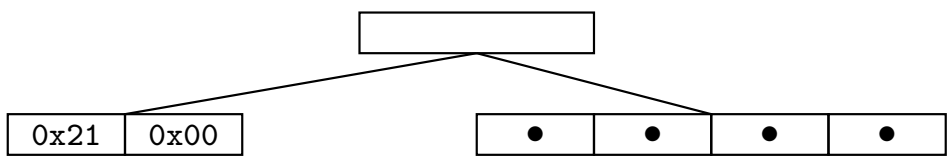
Overview of our memory

Our memory consists of three layers. For

```
struct { short x, *p; } s = { 33; &s.x }
```

consider:

- *Memory values* with arrays of bytes as leafs:



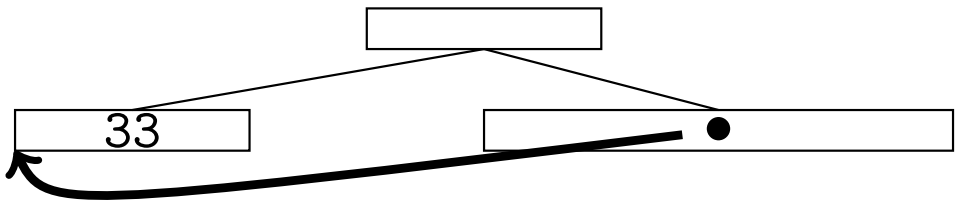
Overview of our memory

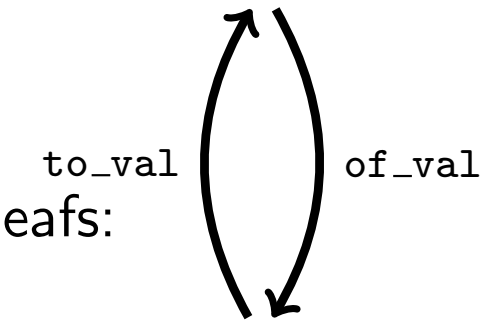
Our memory consists of three layers. For

```
struct { short x, *p; } s = { 33; &s.x }
```

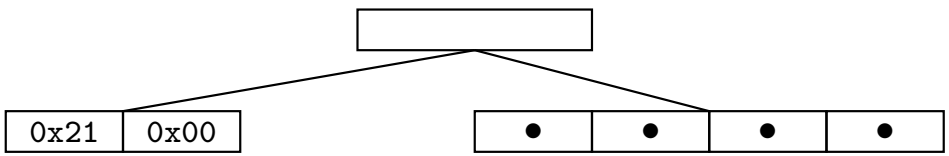
consider:

- *Abstract values* with machine integers and pointers as leafs:





- *Memory values* with arrays of bytes as leafs:



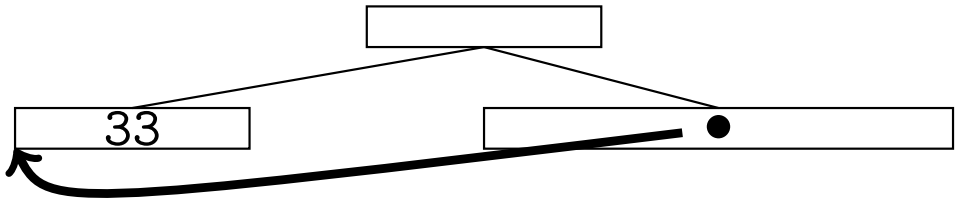
Overview of our memory

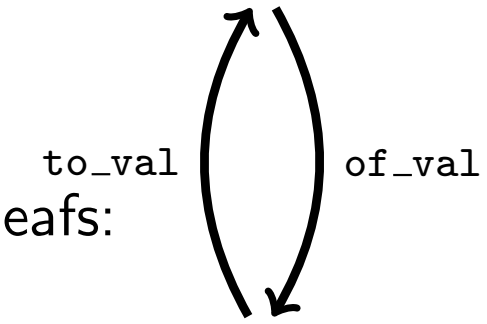
Our memory consists of three layers. For

```
struct { short x, *p; } s = { 33; &s.x }
```

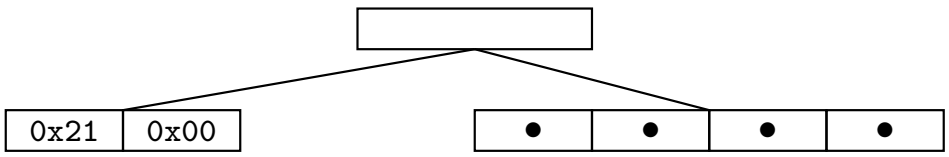
consider:

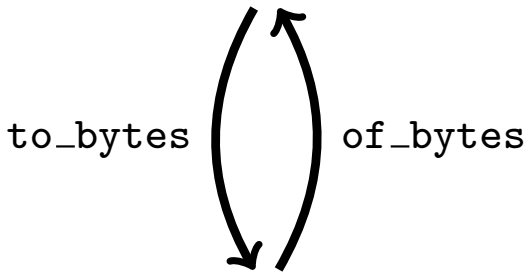
- *Abstract values* with machine integers and pointers as leafs:





- *Memory values* with arrays of bytes as leafs:





- Arrays of bytes:

Bytes and memory values

- *Bytes* are represented symbolically (à la CompCert):

$$b ::= \text{“}\textit{bit sequence}\text{”} \mid (\texttt{ptr}\ p)_i \mid \texttt{undef}$$

- Gives *“the best of both worlds”*: allows bitwise hacking on integers while keeping the memory abstract

Bytes and memory values

- *Bytes* are represented symbolically (à la CompCert):

$$b ::= \text{“bit sequence”} \mid (\texttt{ptr}\ p)_i \mid \texttt{undef}$$

- Gives *“the best of both worlds”*: allows bitwise hacking on integers while keeping the memory abstract
- *Memory values* are defined as:

$$\begin{aligned} w ::= {} & \texttt{base}_{\tau_\mathsf{b}}\ \vec{b} \mid \texttt{array}_\tau\ \vec{w} \\ & \mid \texttt{struct}_s\ \vec{w} \mid \texttt{union}_s\ (i, w) \mid \overline{\texttt{union}_s}\ \vec{b} \end{aligned}$$

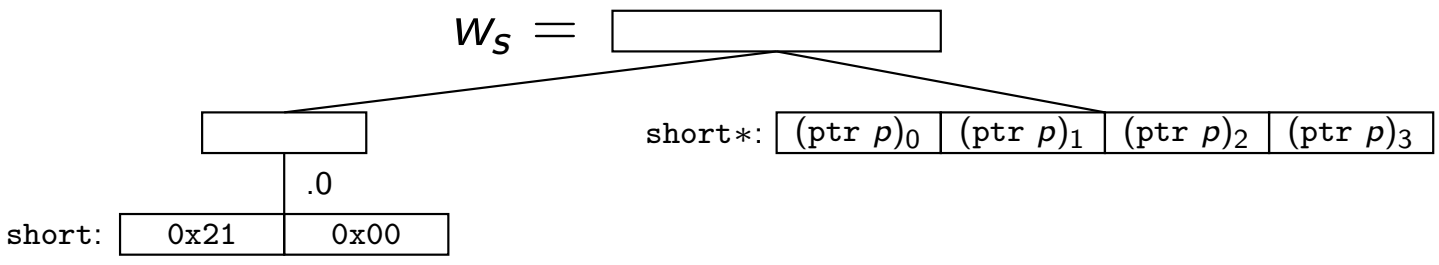
- With corresponding typing judgments

Example

Consider:

```
struct T {
  union U { short x; int y; } u;
  short *p;
} s = { { .x = 33 }; &s.u.x + 1 }
```

As a picture:

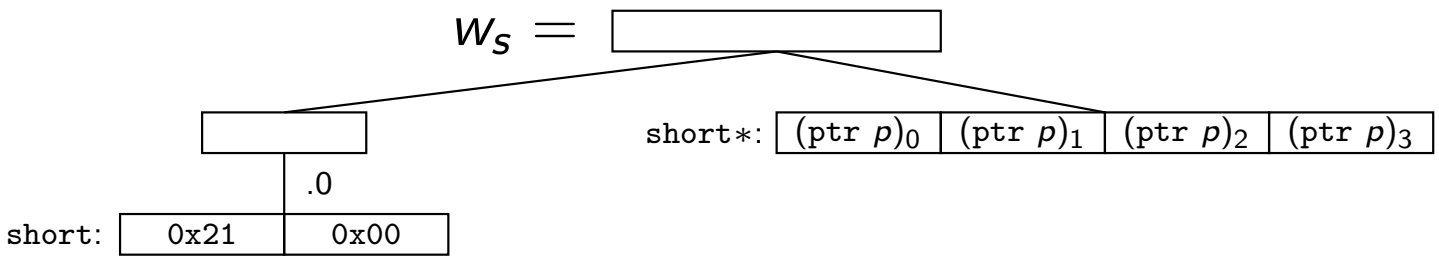


Example

Consider:

```
struct T {
  union U { short x; int y; } u;
  short *p;
} s = { { .x = 33 }; &s.u.x + 1 }
```

As a picture:



Here we have:

- to_bytes $w_s =$

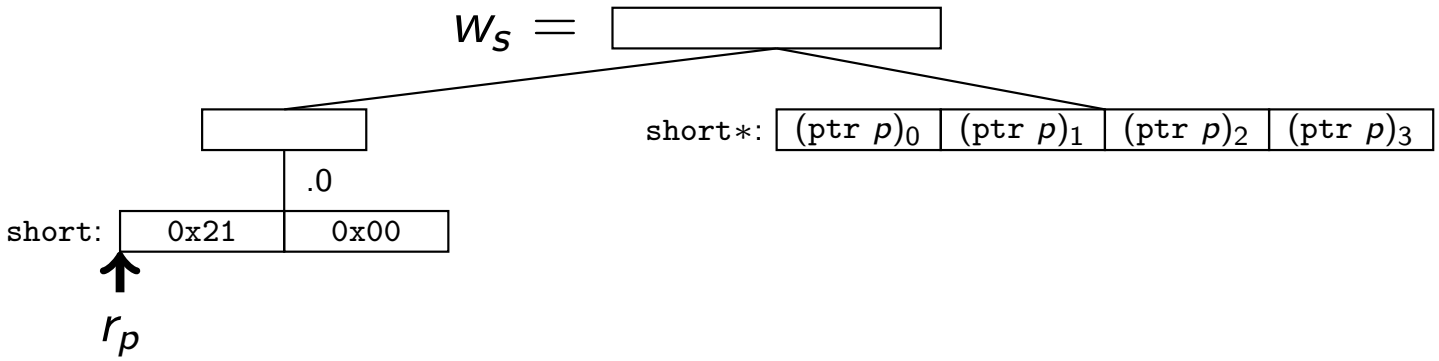
0x21	0x00	undef	undef	(ptr p)$_0$	(ptr p)$_1$	(ptr p)$_2$	(ptr p)$_3$

Example

Consider:

```
struct T {
  union U { short x; int y; } u;
  short *p;
} s = { { .x = 33 }; &s.u.x + 1 }
```

As a picture:



Here we have:

- to_bytes $w_s =$

0x21	0x00	undef	undef	(ptr p)$_0$	(ptr p)$_1$	(ptr p)$_2$	(ptr p)$_3$

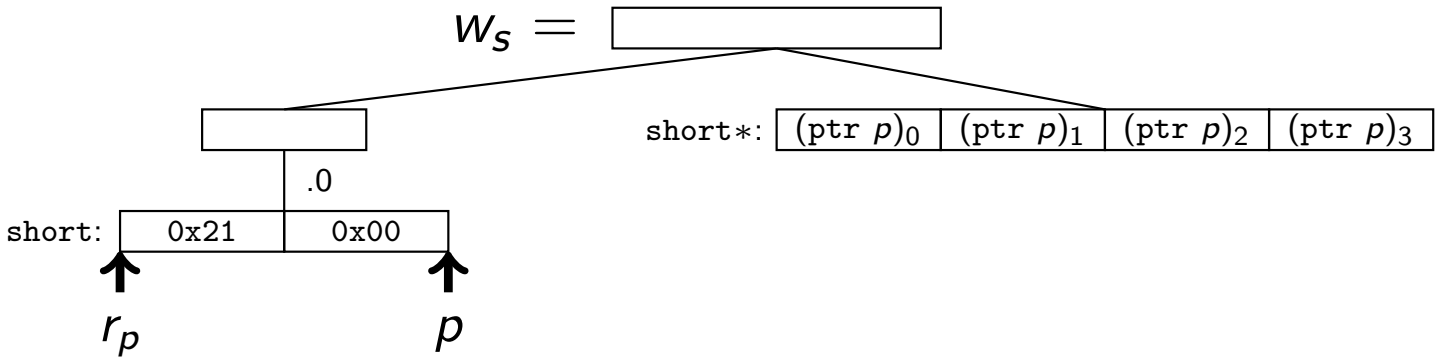
- $r_p = \mathtt{top} \overset{\mathtt{s}}{\rightsquigarrow} 0 \overset{\mathtt{u}}{\rightsquigarrow} 0,$

Example

Consider:

```
struct T {
  union U { short x; int y; } u;
  short *p;
} s = { { .x = 33 }; &s.u.x + 1 }
```

As a picture:



Here we have:

- to_bytes $w_s =$

0x21	0x00	undef	undef	(ptr p)$_0$	(ptr p)$_1$	(ptr p)$_2$	(ptr p)$_3$

- $r_p = \mathtt{top} \overset{\mathtt{s}}{\rightsquigarrow} 0 \overset{\mathtt{u}}{\rightsquigarrow} 0,\ p = (\mathtt{short}, s, r_p, 1)$

Memory values to bytes?

Let us reconsider memory values:

$$w ::= \mathtt{base}_{\tau_\mathsf{b}}\ \vec{b} \mid \mathtt{array}_{\tau}\ \vec{w}$$
$$\mid \mathtt{struct}_{s}\ \vec{w} \mid \mathtt{union}_{s}\ (i, w) \mid \overline{\mathtt{union}_{s}}\ \vec{b}$$

How to do the conversion `of_bytes`?

Memory values to bytes?

Let us reconsider memory values:

$$w ::= \mathtt{base}_{\tau_b}\ \vec{b} \mid \mathtt{array}_{\tau}\ \vec{w}$$
$$\mid \mathtt{struct}_s\ \vec{w} \mid \mathtt{union}_s\ (i, w) \mid \overline{\mathtt{union}_s}\ \vec{b}$$

How to do the conversion `of_bytes`?

- Seems impossible
- Given bytes of

  ```
  union U { int x; int y; }
  ```

  how to know which variant to choose?

Memory values to bytes?

Let us reconsider memory values:

$$w ::= \texttt{base}_{\tau_\texttt{b}}\ \vec{b} \mid \texttt{array}_{\tau}\ \vec{w} \mid \texttt{struct}_{s}\ \vec{w} \mid \texttt{union}_{s}\ (i, w) \mid \overline{\texttt{union}_{s}}\ \vec{b}$$

How to do the conversion `of_bytes`?

- Seems impossible
- Given bytes of

  ```
  union U { int x; int y; }
  ```

  how to know which variant to choose?

Solution: postpone this choice by storing it as a $\overline{\texttt{union}_{\texttt{U}}}\ \vec{b}$ ... and change it into a $\texttt{union}_{\texttt{U}}\ (i, w)$ node on a lookup

Memory values to bytes?

Let us reconsider memory values:

$$w ::= \texttt{base}_{\tau_\text{b}}\ \vec{b} \mid \texttt{array}_{\tau}\ \vec{w} \mid \texttt{struct}_{s}\ \vec{w} \mid \texttt{union}_{s}\ (i, w) \mid \overline{\texttt{union}_{s}}\ \vec{b}$$

How to do the conversion `of_bytes`?

- Seems impossible
- Given bytes of

```
union U { int x; int y; }
```

  how to know which variant to choose?

Solution: postpone this choice by storing it as a $\overline{\texttt{union}_\text{U}}\ \vec{b}$ ... and change it into a $\texttt{union}_\text{U}\ (i, w)$ node on a lookup

Hard part: dealing with this choice in *abstract values* and the various operations

Formalization in Coq

Type theory is ideal for the combination programming/proving

- *The devil is in the details*, Coq is extremely useful for debugging of definitions
- Useful to prove meta-theoretical properties
  e.g. memory embeddings and commuting diagrams
- Use of type classes for parametrization by machine integers
- Use of type classes for overloading of notations
- Light use of dependent types to get better properties on pseudo terms *e.g.* ranges of pointers
- 7.500 lines of code for the memory model out of 28.000

Future research

- Floating point numbers
- Variable length arrays
- The `const`, `volatile` and `restrict` qualifier
- Integration into our operational semantics
  ... and make it (reasonably efficiently) executable
- Memory injections à la CompCert
- Correspondence with CompCert
- Integration into our axiomatic semantics
- Verification Condition generator in Coq

Questions

Sources: see `http://robbertkrebbers.nl/research/ch2o/`

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`
  We have:
  - $p = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} i \rightsquigarrow r_p, i_p)$
  - $q = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} j \rightsquigarrow r_q, i_q)$ with $i \neq j$

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`
  We have:
  - $p = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} i \rightsquigarrow r_p, i_p)$
  - $q = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} j \rightsquigarrow r_q, i_q)$ with $i \neq j$

  Four cases:
  1. $m(z, r) = \texttt{union}_s\ (i, v)$: Now `*q = 3.14` fails

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`
  We have:
  - $p = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} i \rightsquigarrow r_p, i_p)$
  - $q = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} j \rightsquigarrow r_q, i_q)$ with $i \neq j$

  Four cases:
  2. $m(z, r) = \texttt{union}_s\ (j, v)$: Now `*p` fails

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`
  We have:
  - $p = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} i \rightsquigarrow r_p, i_p)$
  - $q = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} j \rightsquigarrow r_q, i_q)$ with $i \neq j$

  Four cases:
  3. $m(z, r) = \texttt{union}_s\ (k, v)$ with $k \notin \{i, j\}$: Now `*p` fails

Aliasing optimizations

Is the optimization `h` $\rightarrow$ `h_opt` correct?

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x; }
int h_opt(int *p, float *q) {
  *q = 3.14; return (*p); }
```

- If $p \perp q$, then `h_opt` is correct
- If not $p \perp q$ we show that undefined behavior occurred in `h`
  We have:
  - $p = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} i \rightsquigarrow r_p, i_p)$
  - $q = (\texttt{int}, z, r \overset{\texttt{u}}{\rightsquigarrow} j \rightsquigarrow r_q, i_q)$ with $i \neq j$

  Four cases:

  4. $m(z, r) = \overline{\texttt{union}_s}\ \vec{b}$:
     Now `*p` changes the memory to $m(z, r) = \texttt{union}_s\ (i, v)$
     After that, `*q = 3.14` fails