Pointer level analysis

Shuchang Zhou

2009-5-5

Motivation

- Proving fields of a union in fact not interfering each other
- Vpr's
  - struct s_heap {
    - …
  - {int prev_node;
  - struct s_heap *next;} u; …
  - };

When will they interfere?

- Suppose
  - t.u.prev_node = 3;
  - p = t.u.next;

When will they interfere?

- Suppose
  - t.u.prev_node = 3;
  - p = t.u.next;
  - **One field is written when the other field is alive.**

Liveness analysis?

Liveness analysis?

- Very difficult as heap data is involved
- Require full fledged tools. May employ decision procedure that have exponential time or even no guarantee to terminate.

Type safety

- Observe they are of different types.
  - {int prev_node;
  - struct s_heap *next;}
- Assume the original program is type safe. Then they cannot interfere without explicit type casts.

Type safety

- Observe they are of different types.
  - {int prev_node;
  - struct s_heap *next;}
- Assume the original program is type safe. Then they cannot interfere without explicit type casts.

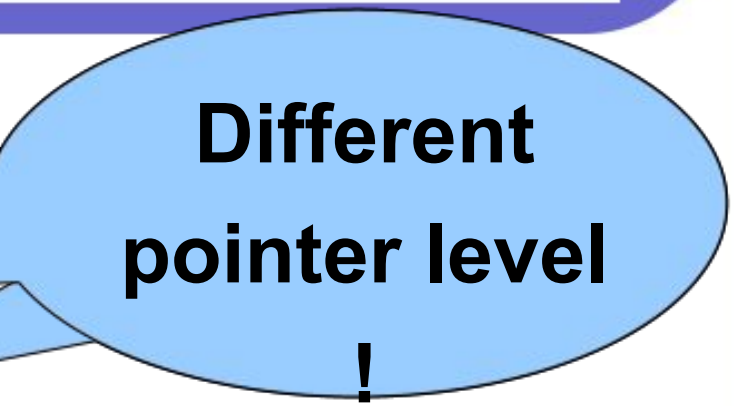
Very strong assumption!

Type safety

- Many common idioms in C violates type-safety
- char* p = (char*) malloc (…
- int i = 352;
- char c = i; // to get the lower 8-bit.

Type safety

- Observe they are of diffe
  - {int prev_node;
  - struct s_heap *next;}
- Assume the original program is type safe. Then they cannot interfere without explicit type casts.



Prove the type safety

- Don’t trust original types in the source, analyze by our own.
- Only care about “ranks”, i.e., the pointer levels. Intuitively:

Prove the type safety

- Don’t trust original types in the source, analyze by our own.
- Only care about “ranks”, i.e., the pointer levels. Intuitively:
  - a[i] = 3; // “i” is of rank 0 as it is used as an index.
  - Int a[20]; // “a” is of rank 1
  - b = c ; // “b” has the same rank as “c”

Prove the type safety

- Don't trust original types in the source, analyze by our own
- Only care about levels. Intuitively:
  - a[i] = 3; // "i" is of rank as it ed as an index.
  - Int a[20]; // "a" is of rank 1
  - b = c ; // "b" has the same rank as "c"

But we don't want to trust original types!

Prove the type safety

- Rules:
  - a=b => R(a) = R(b)
  - a.prev_node = b => R(prev_node) = R(b)
  - seeing a[i] => R(i) = 0
  - a->u.prev_node =>

    “a” is of type struct s_heap *

Prove the type safety

- Rules:
  - a=b => R(a) = R(b)
  - a.prev_node = b => R
  - seeing a[i] => R
  - a->u.prev_node =>

    “a” is of type struct s_heap *

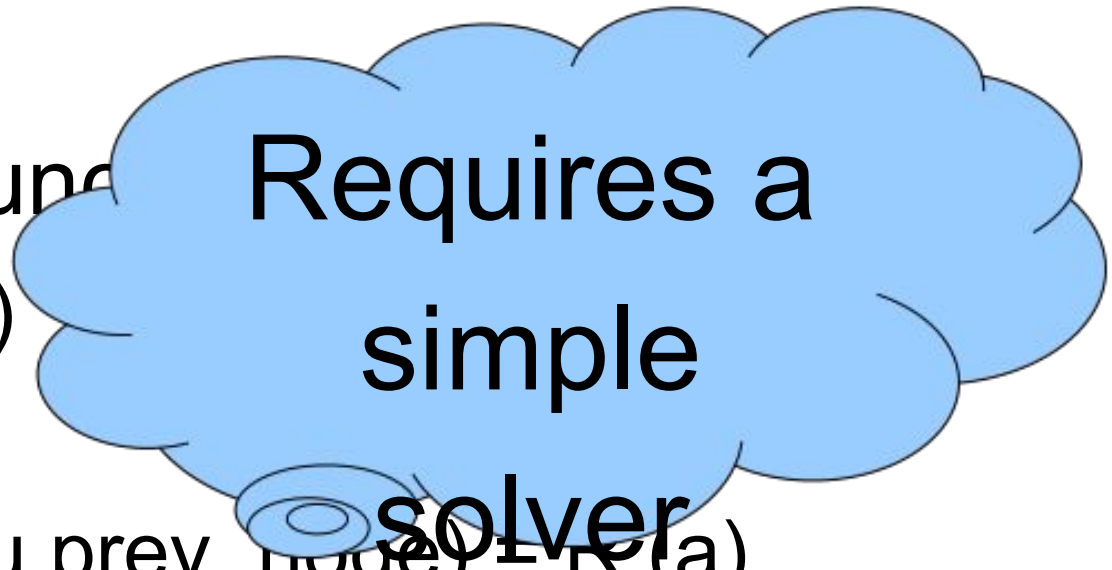
We guarantee that we don’t confuse prev_node field from different struct/unions

Prove the type safety

- We get a bunch of equations.
  - R(a) = R(b)
  - R(i) = 0
  - R(s_heap.u.prev_node) = R (a)

Prove the type safety

- We get a bunc
  - R(a) = R(b)
  - R(i) = 0
  - R(s_heap.u.prev_node) = R(a)



Prove the type safety

- The solver of equations

```
type elem =
   | EVarinfo of varinfo
   | EField of fieldinfo
```

```
type 'a equation=
   | Equal of 'a * 'a *int
   | Val of 'a * int
```

  - Elim an variable each pass

Prove the type safety

- The solver of equations

```
type elem =
   | EVarinfo of varinfo
   | EField of fieldinfo

type 'a equation=
   | Equal of 'a * 'a *int
   | Val of 'a * int
```

For variables

For fields

  - Elim an variable each pass

Prove the type safety

- The solver of equations

```
type elem =
   | EVarinfo of varinfo
   | EField of fieldinfo

type 'a equation=
   | Equal of 'a * 'a *int
   | Val of 'a * int
```

R(a) = R(b) + n

R(a) = n

  - Elim an variable each pass

Prove the type safety

- The solver of equations

```
(*post: occurrences of s is elim'ed *)
let applyVal s i e = match e with
  | Val(s',i') ->
    if (neq s' s || i=i') then return e else
      raise Inconsistent
  | Equal(s',s2',i') ->
    if eq s' s then return (Val(s2',i-i')) else
      if eq s2' s then return (Val(s',i'+i)) else
        return e
```

Prove the type safety

- The solver of equations

Introduce
R(s) = i

```
(*post: occurrences of s is elim'ed *)
let applyVal s i e = match e with
  | Val(s',i') ->
    if (neq s' s || i=i') then return e else
      raise Inconsistent
  | Equal(s',s2',i') ->
    if eq s' s then return (Val(s2',i-i')) else
      if eq s2' s then return (Val(s',i'+i)) else
        return e
```

Prove the type safety

- The solver of equations

Introduce

R(s) = R(s2) + i

```
(*pre: e is processed by simp
(*post: occurrences of s is elim'ed *)
let applyEq s s2 i e = match e with
  | Val(s',i') ->
    if eq s s' then return (Val(s2,i'-i)) else return e
  | Equal(s',s2',i') ->
    if eq s s' then simp (Equal(s2,s2',i'-i)) else
    if eq s s2' then simp (Equal(s',s2,i+i')) else
      return e
```

Consequence of type safety

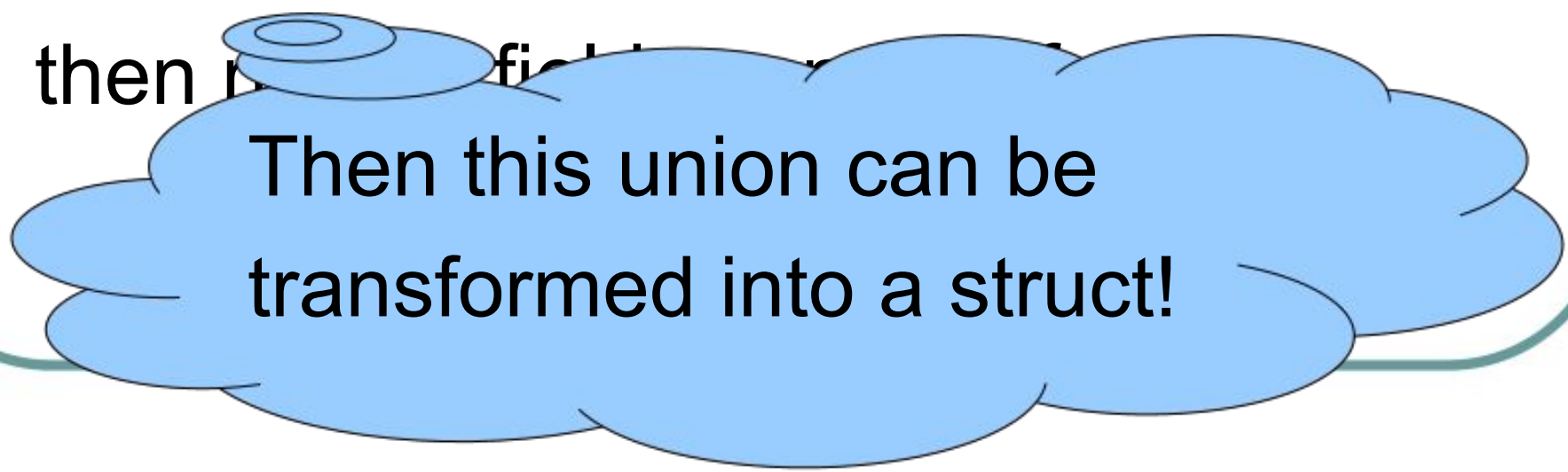
- If no Inconsistent exception is raised, then we prove type safety with respect to pointer level.

Consequence of type safety

- If no Inconsistent exception is raised, then we prove type safety with respect to pointer level.
- Under type safety just proved, when no two fields of a union has the same rank, then no two fields can interfere.

Consequence of type safety

- If no Inconsistent exception is raised, then we prove type safety with respect to pointer level.
- Under type safety just proved, when no two fields of a union has the same rank, then



Character of the analysis

- Flow-insesitive
  - May be conservative
- Elimination-based
  - Faster than iteration-based as equations are simple
- Partition of pointers.
  - Serve as first step for pointer analysis?

Restriction

- Unable to analyze “magic number” pointers.
  - int i = 0xffffff32;
  - char j = *((char*) i) ;

Questions?

- Thank you!

Consequence of type safety

- Under type safety just proved, when no two fields of a union has the same rank, then no two fields can interfere. Then this union can be transformed into a struct.
- We limit us to transform only locally defined unions.