

Using type inference to discover
interesting properties about programs:

Lackwit

Amer Diwan

Goals of the paper

- Use type inference to find bugs in programs
- Idea:
 - Refine programming language types and compute them by type inference (e.g., int^1 , int^2)
 - If two unrelated uses have the same refined type then it may suggest a problem

Why are we reading this paper?

- Another interesting use of type inference: finding bugs
- A polymorphic or context-sensitive type inference algorithm
 - SML's algorithm is polymorphic
 - Steensgaard's algorithm is not

Program understanding using type inference: A simple example

- ```
note_score(int student_id, int score) {
 scores[student_id] += score;
}
```
- ```
main() {
    astudent = read_int();
    ascore = read_int();
    note_score(ascore, astudent);
    ...
}
```

Continuing with the example

- What code constrains the representation of "astudent"?
- `note_score(int student_id, int score) {
 scores[student_id] += score;
}`
- `main() {
 astudent = read_int();
 ascore = read_int();
 note_score(ascore, astudent);
}`

Monomorphism in type inference

- E.g., Steensgaard's pointer analysis
- **Parameter types are not polymorphic**
- E.g.,

```
int *f(int *p) {  
    return p;  
}  
int A, B;  
t = f(&A);  
u = f(&B);
```

Types:

p: $\tau_1 = \text{ref}(\tau_2, _)$

A: $\tau_2 = \text{ref}(\perp, \perp)$

B: τ_2

t, u: τ_1

But t and u should point to different things...

Problems with monomorphism

- In monomorphic information from different calling contexts gets merged together
- SML type inference is polymorphic:
fun id(v: 'a): 'a = v
- Lackwit is also polymorphic

Polymorphic inference

- Type inference rules are the “same” but how one evaluates a call is different
 - At each call the polymorphic type parameters are instantiated and unified with the corresponding actual arguments

A bigger example

```
intT1 x;
intT2 p1;
void f(intT3 a, intT4 b, intT5 *c, intT6
    *d) {
    x = a;
    *c = *d;
}

void g(intT7 *q, intT8 *r, intT9 *s) {
    intT10 t1 = 2;
    intT11 c1 = 3;
    intT12 c2 = 4;
    intT13 p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
```

Example continued

Globals must have fixed type, things with which you compute must have type "n" => T1->n, T2->z

```
intn x;
intz p1;
void f(intT3 a, intT4 b, intT5 *c, intT6
    *d) {
    x = a;
    *c = *d;
}

void g(intT7 *q, intT8 *r, intT9 *s) {
    intT10 t1 = 2;
    intT11 c1 = 3;
    intT12 c2 = 4;
    intT13 p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
```

Example continued

$e1: \tau \quad e2: \tau$ applying it to $x=a, *c=*d, p=p1$

$e1 = e2: \tau$

```

intn x;
intz p1;
void f(intn a, intT4 b, intB *c, intB
    *d) {
    x = a,
    *c = *d;
}

void g(intT7 *q, intT8 *r, intT9 *s) {
    intT10 t1 = 2;
    intT11 c1 = 3;
    intT12 c2 = 4;
    intz p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}

```

Example continued

$e1: (\tau1, \tau2) \rightarrow \tau, e2: \tau1, e3: \tau2$ apply it to first call

$e1(e2, e3): \tau$

```

intn x;
intz p1;
void f(intn a, intX b, intY *c, intY *d) {...}

void g(intT7 *q, intT8 *r, intT9 *s) {
    intT10 t1 = 2;
    intT11 c1 = 3;
    intz p;
    f(c1, p, &t1, q);}

```

- Make a copy of f's signature using fresh type variables:
void f(intⁿ a, int^X b, int^Y *c, int^Y *d)
- Unify types: (n, T11), (X, T4), (T10, Y), (T7, Y)
 - T11 becomes n, T11 and T7 become Y

Insight: how polymorphic type inference works

- Key idea: unify actual types with a copy of the function type
 - Avoids constraining polymorphic types of function parameters and returns
 - Need to analyze a function only once after which use its polymorphic type

After applying to first call

```
inta x;  
intz p1;  
void f(inta a, intT4 b, intB *c, intB  
    *d) {  
    x = a;  
    *c = *d;  
}
```

```
void g(intY *q, intT8 *r, intT9 *s) {  
    intY t1 = 2;  
    inta c1 = 3;  
    intT12 c2 = 4;  
    intz p;  
    p = p1;  
    x++;  
    f(c1, p, &t1, q);  
    f(c2, c2, r, s);  
}
```

Example continued

$e1: (\tau1, \tau2) \rightarrow \tau, e2: \tau1, e3: \tau2$ apply it to second call

$e1(e2, e3): \tau$

```
intn x;  
intz p1;  
void f(intn a, intT4 b, intB *c, intB  
    *d) {  
    x = a;  
    *c = *d;  
}
```

```
void g(intY *q, intX *r, intX *s) {  
    intY t1 = 2;  
    intn c1 = 3;  
    intn c2 = 4;  
    intz p;  
    p = p1;  
    x++;  
    f(c1, p, &t1, q);  
    f(c2, c2, r, s);  
}
```

Discussion

- What kinds of information can we compute using this technique and how?
 - what affects and what is affected by a variable
 - what pointers may never be deallocated
 - pointer aliases
 - ...?

Next topic

- Using types for optimization
- Reading: Diwan, McKinley, Moss (from web page)