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.,

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

Types:

- p: τ1 = ref(τ2, _)
- A: τ2 = ref(⊥, ⊥)
- B: τ2
- t, u: τ1

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

```c
int T1 x;
int T2 p1;
void f(int aT3, int T4 b, int T5 *c, int T6 *d) {
    x = a;
    *c = *d;
}
void g(int T7 *q, int T8 *r, int T9 *s) {
    int T10 t1 = 2;
    int T11 c1 = 3;
    int T12 c2 = 4;
    int T13 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

```c
int x;
int p1;
void f(int aT3, int T4 b, int T5 *c, int T6 *d) {
    x = a;
    *c = *d;
}
void g(int T7 *q, int T8 *r, int T9 *s) {
    int T10 t1 = 2;
    int T11 c1 = 3;
    int T12 c2 = 4;
    int T13 p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
Example continued

\[
e_1: \tau \quad e_1: \tau \quad \text{applying it to } x=a, \; *c=^*d, \; p=p_1
\]
\[
e_1 = e_2: \tau
\]

```c
int a x;
int p1;
void f(int a, int b, int c, int d) {
    x = a;
    *c = *d;
}
```

```c
void g(int q, int r, int s) {
    t1 = 2;
    c1 = 3;
    c2 = 4;
    p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
```

Example continued

\[
e_1: (\tau_1, \tau_2) -> \tau, \quad e_2: \tau_1, \quad e_3: \tau_2 \quad e_1: \tau \quad \text{apply it to first call }
\]
\[
e_1(e_2, e_3): \tau
\]

```c
int x;
int p1;
void f(int a, int b, int c, int d) {
    x = a;
    *c = *d;
}
```

```c
void g(int *q, int *r, int *s) {
    t1 = 2;
    c1 = 3;
    c2 = 4;
    p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
```
Example continued

\[
e_1 : (\tau_1, \tau_2) \rightarrow \tau, \quad e_2 : \tau_1, \quad e_3 : \tau_2 \quad e_1 : \tau
\]

\[
e_1(e_2, e_3) : \tau
\]

apply it to second call

\[
f(c_1, \texttt{p}, \&\texttt{t1}, \texttt{q})
\]

\[
f(c_2, c_2, \texttt{r}, \texttt{s})
\]

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)