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"?

```cpp
• note_score(int student_id, int score) {
  scores[student_id] += score;
}
• main() {
  asudent = 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.,

```cpp
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: \( \tau_2 \)
- t, u: \( \tau_1 \)

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:
  ```plaintext
  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 T3 a, 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 T1 x;
int T2 p1;
void f(int T3 a, 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

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

e1 = e2: \tau

\begin{verbatim}
int\# x;
int\# p1;
void f(int\# a, int\#4 b, int\# c, int\# d) {
    x = a;
    *c = *d;
}
\end{verbatim}

\begin{verbatim}
void g(int\#7 q, int\#8 r, int\#9 s) {
    int\#10 t1 = 2;
    int\#11 c1 = 3;
    int\#12 c2 = 4;
    int\# p;
    p = p1;
    x++;  
f(c1, p, &t1, q);
f(c2, c2, r, s);
}
\end{verbatim}

\hspace{1cm}e1: (\tau1, \tau2)\rightarrow \tau, e2: \tau1, e3: \tau2\quad \text{apply it to first call}

e1(e2, e3): \tau

\begin{verbatim}
int\# x;
int\# p1;
void f(int\# a, int\#4 b, int\# c, int\# d) {...}
\end{verbatim}

\begin{verbatim}
void g(int\#7 q, int\#8 r, int\#9 s) {
    int\#10 t1 = 2;
    int\#11 c1 = 3;
    int\#12 c2 = 4;
    int\# p;
    p = p1;
    f(c1, p, &t1, q);
}
\end{verbatim}

• Make a copy of f’s signature using fresh type variables:
  void f(int\# a, int\# b, int\# c, int\# 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

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

void g(int *q, int *r, int *s) {
    int t1 = 2;
    int c1 = 3;
    int t12 c2 = 4;
    int p;
    p = p1;
    x++;
    f(c1, p, &t1, q);
    f(c2, c2, r, s);
}
```
Example continued

e1: (τ1, τ2)→ τ, e2: τ1, e3: τ2

e1(e2, e3): τ

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

void g(int q, int r, int s) {
    int t1 = 2;
    int c1 = 3;
    int c2 = 4;
    int 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)