Static Program Analysis
Part 2 – type analysis and unification

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University
Type errors

• Reasonable restrictions on operations:
  – arithmetic operators apply only to integers
  – comparisons apply only to like values
  – only integers can be input and output
  – conditions must be integers
  – only functions can be called
  – the * operator applies only to pointers
  – field lookup can only be performed on records

• Violations result in runtime errors
Type checking

• Can type errors occur during runtime?
• This is interesting, hence instantly undecidable

• Instead, we use conservative approximation
  – a program is typable if it satisfies some type constraints
  – these are systematically derived from the syntax tree
  – if typable, then no runtime errors occur
  – but some programs will be unfairly rejected (slack)

• What we shall see next is the essence of the Damas–Hindley–Milner type inference technique, which forms the basis of the type systems of e.g. ML, OCaml, and Haskell
Typability

- no type errors
- typable
- slack
Fighting slack

• Make the type checker a bit more clever:

• An eternal struggle
Fighting slack

- Make the type checker a bit more clever:

- An eternal struggle
- And a great source of publications
Be careful out there

• The type checker may be unsound:

• Example: covariant arrays in Java
  – a deliberate pragmatic choice
Generating and solving constraints

\[ p = \text{int} \land q = \text{int} \land \text{alloc} = \text{int} \land x = \phi \land \text{foo} = \phi \land n = \text{int} \land \text{main} = () \rightarrow \text{int} \]
Types

• Types describe the possible values:

\[
\begin{align*}
\tau & \rightarrow \text{int} \\
| & \quad \& \tau \\
| & \quad (\tau, \ldots, \tau) \rightarrow \tau \\
| & \quad \{X: \tau, \ldots, X: \tau\}
\end{align*}
\]

• These describe integers, pointers, functions, and records

• Types are *terms* generated by this grammar – example: \((\text{int}, \&\text{int}) \rightarrow \&\&\text{int}\)
Type constraints

• We generate type constraints from an AST:
  – all constraints are equalities
  – they can be solved using a unification algorithm

• Type variables:
  – for each identifier declaration $X$ we have the variable $[X]$
  – for each non-identifier expression $E$ we have the variable $[E]$

• Recall that all identifiers are unique

• The expression $E$ denotes an AST node, not syntax

• (Possible extensions: polymorphism, subtyping, …)
### Generating constraints (1/3)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I$</td>
<td>$\lceil I \rceil = \text{int}$</td>
</tr>
<tr>
<td>$E_1 \text{ op } E_2$</td>
<td>$\lceil E_1 \rceil = \lceil E_2 \rceil = \lceil E_1 \text{ op } E_2 \rceil = \text{int}$</td>
</tr>
<tr>
<td>$E_1 == E_2$</td>
<td>$\lceil E_1 \rceil = \lceil E_2 \rceil \land \lceil E_1 == E_2 \rceil = \text{int}$</td>
</tr>
<tr>
<td>input</td>
<td>$\lceil \text{input} \rceil = \text{int}$</td>
</tr>
<tr>
<td>$X = E$</td>
<td>$\lceil X \rceil = \lceil E \rceil$</td>
</tr>
<tr>
<td>output $E$</td>
<td>$\lceil E \rceil = \text{int}$</td>
</tr>
<tr>
<td>if ($E$) {S}</td>
<td>$\lceil E \rceil = \text{int}$</td>
</tr>
<tr>
<td>if ($E$) {S$_1$} else {S$_2$}</td>
<td>$\lceil E \rceil = \text{int}$</td>
</tr>
<tr>
<td>while ($E$) {S}</td>
<td>$\lceil E \rceil = \text{int}$</td>
</tr>
</tbody>
</table>
Generating constraints (2/3)

\[ X(X_1,\ldots,X_n) \{ \ldots \text{return } E; \} : \]
\[ \{X\} = ([X_1], \ldots, [X_n]) \rightarrow [E] \]

\[ E(E_1, \ldots, E_n) : \]
\[ \{E\} = ([E_1], \ldots, [E_n]) \rightarrow [E(E_1, \ldots, E_n)] \]

\[ \text{alloc } E : \]
\[ \{\text{alloc } E\} = &\{E\} \]

\[ \text{&x} : \]
\[ \{\text{&x}\} = &\{X\} \]

\[ \text{null} : \]
\[ \{\text{null}\} = \&\alpha \quad \text{(each } \alpha \text{ is a fresh type variable)} \]

\[ \ast E : \]
\[ \{E\} = \&\{\ast E\} \]

\[ \ast X = E : \]
\[ \{X\} = \&\{E\} \]
Generating constraints (3/3)

\[\{X_1 : E_1, \ldots, X_n : E_n\}:\]
\[\llbracket \{X_1 : E_1, \ldots, X_n : E_n\} \rrbracket = \{X_1 : \llbracket E_1 \rrbracket, \ldots, X_n : \llbracket E_n \rrbracket\}\]
\[E . X:\]
\[\llbracket E \rrbracket = \{\ldots, X : \llbracket E . X \rrbracket, \ldots\}\]

This is the idea, but not directly expressible in our language of types
Generating constraints (3/3)

Let \( \{f_1, f_2, \ldots, f_m\} \) be the set of field names that appear in the program

\[
\{X_1: E_1, \ldots, X_n: E_n\}: \quad \llbracket \{X_1: E_1, \ldots, X_n: E_n\} \rrbracket = \{f_1: \gamma_1, \ldots, f_m: \gamma_m\}
\
\text{where } \gamma_i = \begin{cases} 
\llbracket E_j \rrbracket & \text{if } f_i = X_j \text{ for some } j \\
\alpha_i & \text{otherwise}
\end{cases}
\
E.X: \quad \llbracket E \rrbracket = \{f_1: \gamma_1, \ldots, f_m: \gamma_m\}
\
\text{where } \gamma_i = \begin{cases} 
\llbracket E.X \rrbracket & \text{if } f_i = X \\
\alpha_i & \text{otherwise}
\end{cases}
Exercise

main() {
  var x, y, z;
  x = input;
  y = alloc 8;
  *y = x;
  z = *y;
  return x;
}

• Generate and solve the constraints
• Then try with y = alloc 8 replaced by y = 42
• Also try with the Scala implementation (when it’s completed)
General terms

Constructor symbols:

- 0-ary: a, b, c
- 1-ary: d, e
- 2-ary: f, g, h
- 3-ary: i, j, k

Terms:

- a
- d(a)
- h(a, g(d(a), b))

Terms with variables:

- f(X, b)
- h(X, g(Y, Z))
The unification problem

• An equality between two terms with variables:

\[ k(X,b,Y) = k(f(Y,Z),Z,d(Z)) \]

• A solution (a unifier) is an assignment from variables to closed terms that makes both sides equal:

\[ X = f(d(b),b) \]
\[ Y = d(b) \]
\[ Z = b \]

Implicit constraint for term equality:
\[ c(t_1,...,t_k) = c(t'_1,...,t'_k) \Rightarrow t_i = t'_i \text{ for all } i \]
Unification errors

• Constructor error:

\[ d(X) = e(X) \]

• Arity error:

\[ a = a(X) \]
The linear unification algorithm

- Paterson and Wegman (1978)
- In time $O(n)$:
  - finds a most general unifier
  - or decides that none exists

- Can be used as a back-end for type checking

- ... but only for finite terms
Recursive data structures

The program

```
var p;
p = alloc null;
*p = p;
```

creates these constraints

```
[null] = &\alpha
[alloc null] = &[null]
[p] = &[alloc null]
[p] = &[p]
```

which have this “recursive solution” for p:

```
[p] = \alpha  \text{ where } \alpha = &\alpha
```
Regular terms

• Infinite but (eventually) repeating:
  – $e(e(e(e(e(...))))))$
  – $d(a,d(a,d(a, ...)))$
  – $f(f(f(...),f(...)),f(f(...),f(...))),f(f(f(...),f(...)),f(f(...),f(...)))$)

• Only finitely many *different* subtrees

• A non-regular term:
  – $f(a,f(d(a),f(d(d(a))),f(d(d(d(a)))),...,)))$
Regular unification

• Huet (1976)
• The unification problem for regular terms can be solved in $O(n \cdot A(n))$ using a union-find algorithm

• $A(n)$ is the inverse Ackermann function:
  – smallest $k$ such that $n \leq \text{Ack}(k,k)$
  – this is never bigger than 5 for any real value of $n$

• See the TIP implementation...
Union-Find

makeset(x) {
    x.parent := x
    x.rank := 0
}

find(x) {
    if x.parent != x
        x.parent := find(x.parent)
    return x.parent
}

union(x, y) {
    xr := find(x)
    yr := find(y)
    if xr = yr
        return
    if xr.rank < yr.rank
        xr.parent := yr
    else
        yr.parent := xr
    if xr.rank = yr.rank
        xr.rank := xr.rank + 1
}
Union-Find (simplified)

makeset(x) {
    x.parent := x
}

find(x) {
    if x.parent != x
        x.parent := find(x.parent)
    return x.parent
}

union(x, y) {
    xr := find(x)
    yr := find(y)
    if xr = yr
        return
        xr.parent := yr
}

Implement ‘unify’ procedure using union and find to unify terms...
Implementation strategy

• Representation of the different kinds of types (including type variables)
• Map from AST nodes to types
• Union-Find
• Traverse AST, generate constraints, unify on the fly
  – report type error if unification fails
  – when unifying a type variable with e.g. a function type, it is useful to pick the function type as representative
  – for outputting solution, assign names to type variables (that are roots), and be careful about recursive types
The complicated function

foo(p, x) {
    var f, q;
    if (*p == 0) {
        f = 1;
    } else {
        q = alloc 0;
        *q = (*p) - 1;
        f = (*p) * (x(q, x));
    }
    return f;
}

main() {
    var n;
    n = input;
    return foo(&n, foo);
}
Generated constraints

```plaintext
[foo] = ([p],[x]) -> [f]
[*p] = int
[1] = int
[1] = int
[p] = &[*p]
[alloc 0] = &[0]
[q] = &[*q]
[f] = [(*p)*(x(q,x))] [x(q,x)] = int
[input] = int
[n] = [input]
[foo] = ([&n],[foo]) -> [foo(&n,foo)]

[*p==0] = int
[f] = [1]
[0] = int
[q] = [alloc 0]
[q] = &[(*p)-1]
[*p] = int
[(*p)*(x(q,x))] = int
[x] = ([q],[x]) -> [x(q,x)]
[main] = () -> [foo(&n,foo)]
[&n] = &[n]
[(*p)-1] = int
[*p] = [0]
```
Here, $\phi$ is the regular type that is the unfolding of $\phi = (\text{\&int}, \phi) \rightarrow \text{\&int}$ which can also be written $\phi = \mu \alpha. (\text{\&int}, \alpha) \rightarrow \text{\&int}$

All other variables are assigned \text{\&int}
Infinitely many solutions

The function

```cpp
poly(x) {
    return *x;
}
```

has type `(\&\alpha) \rightarrow \alpha` for any type \(\alpha\)

(which is not expressible in our current type language)
Recursive and polymorphic types

• Extra notation for recursive and polymorphic types:

\[
\tau \longrightarrow \ldots
\]

| \( \mu \alpha. \tau \) |
| \( \alpha \) |

(not very useful unless we also add polymorphic expansion at calls, but that makes complexity exponential, or even undecidable...)

• Types are (finite) terms generated by this grammar
• \( \mu \alpha. \tau \) is the (potentially recursive) type \( \tau \) where occurrences of \( \alpha \) represent \( \tau \) itself
• \( \alpha \) is a type variable (implicitly universally quantified if not bound by an enclosing \( \mu \))
This never has a type error at runtime – but it is not typable:

\[ \text{int} = [r] = [g] = \&\alpha \]
Other errors

• Not all errors are type errors:
  – dereference of null pointers
  – reading of uninitialized variables
  – division by zero
  – escaping stack cells
(why not?)

• Other kinds of static analysis may catch these

```plaintext
baz() {  
  var x;  
  return &x; 
}

main() {  
  var p;  
  p=baz();  
  *p=1;  
  return *p; 
}
```