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 only applies to pointers
  - field lookup can only be performed on records
  - the fields being accessed are guaranteed to be present
- Violations result in runtime errors
- Note: no type annotations in TIP

Type checking

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

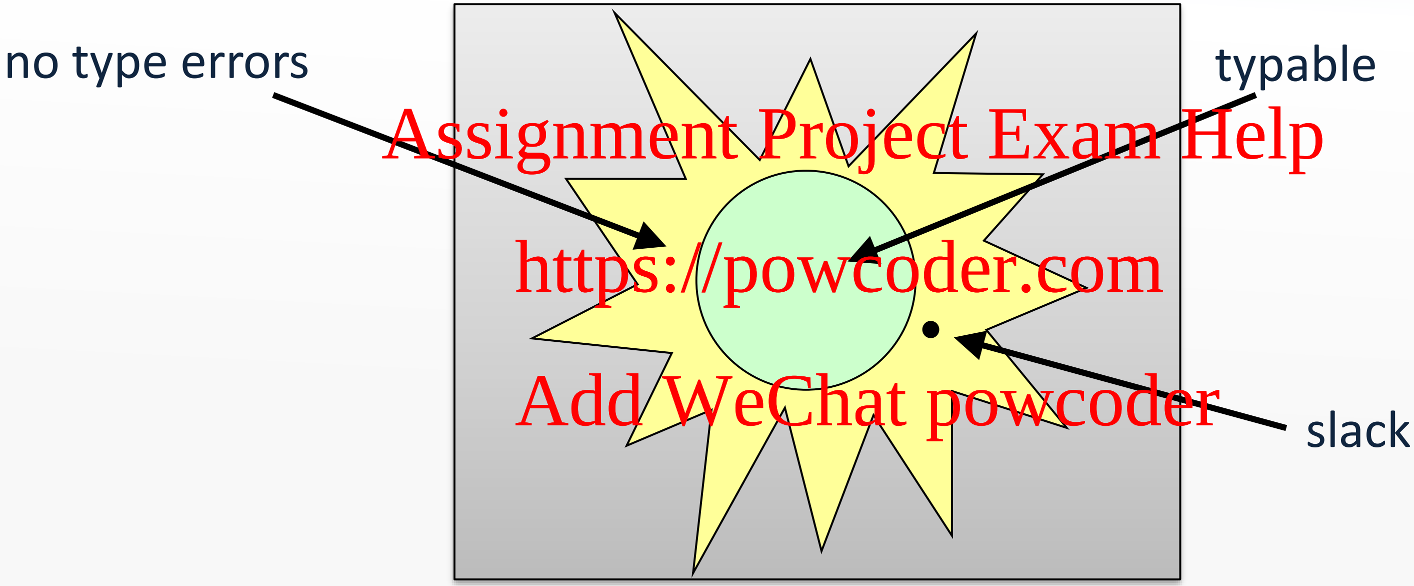
- Instead, we use conservative approximation

  - 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*)

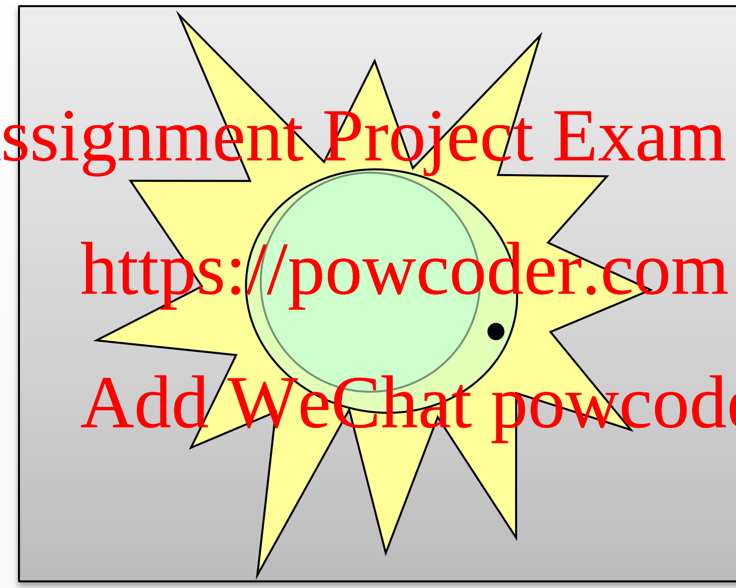
- What we shall see next is the essence of the Damas–Hindley–Milner type inference technique, which forms the basis of the type systems of e.g. ML, OCaml, and Haskell

Typability



Fighting slack

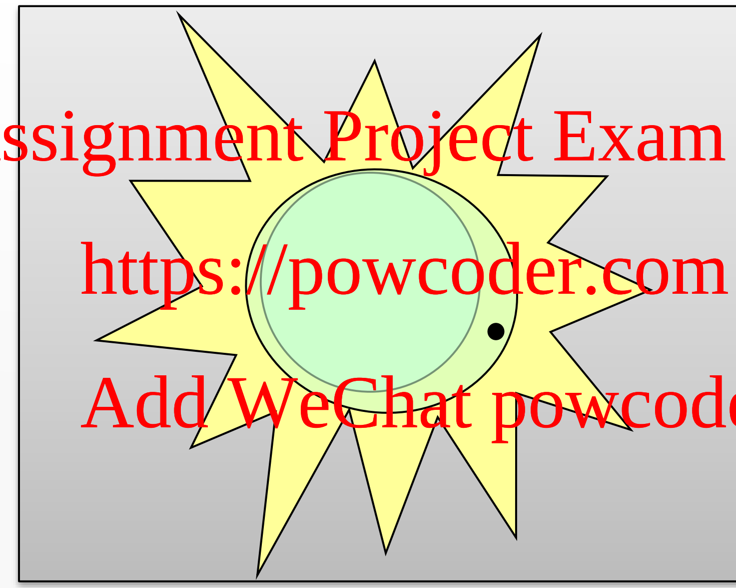
- Make the type checker a bit more clever:



- An eternal struggle

Fighting slack

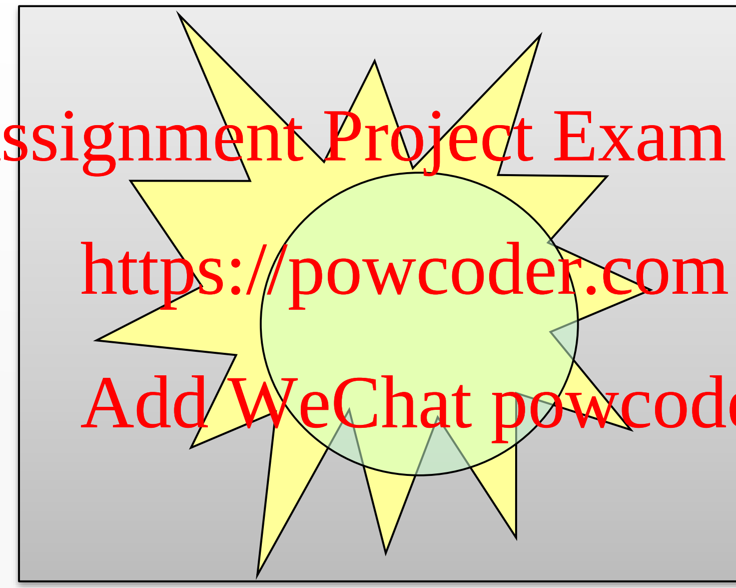
- Make the type checker a bit more clever:



- An eternal struggle
- And a great source of publications

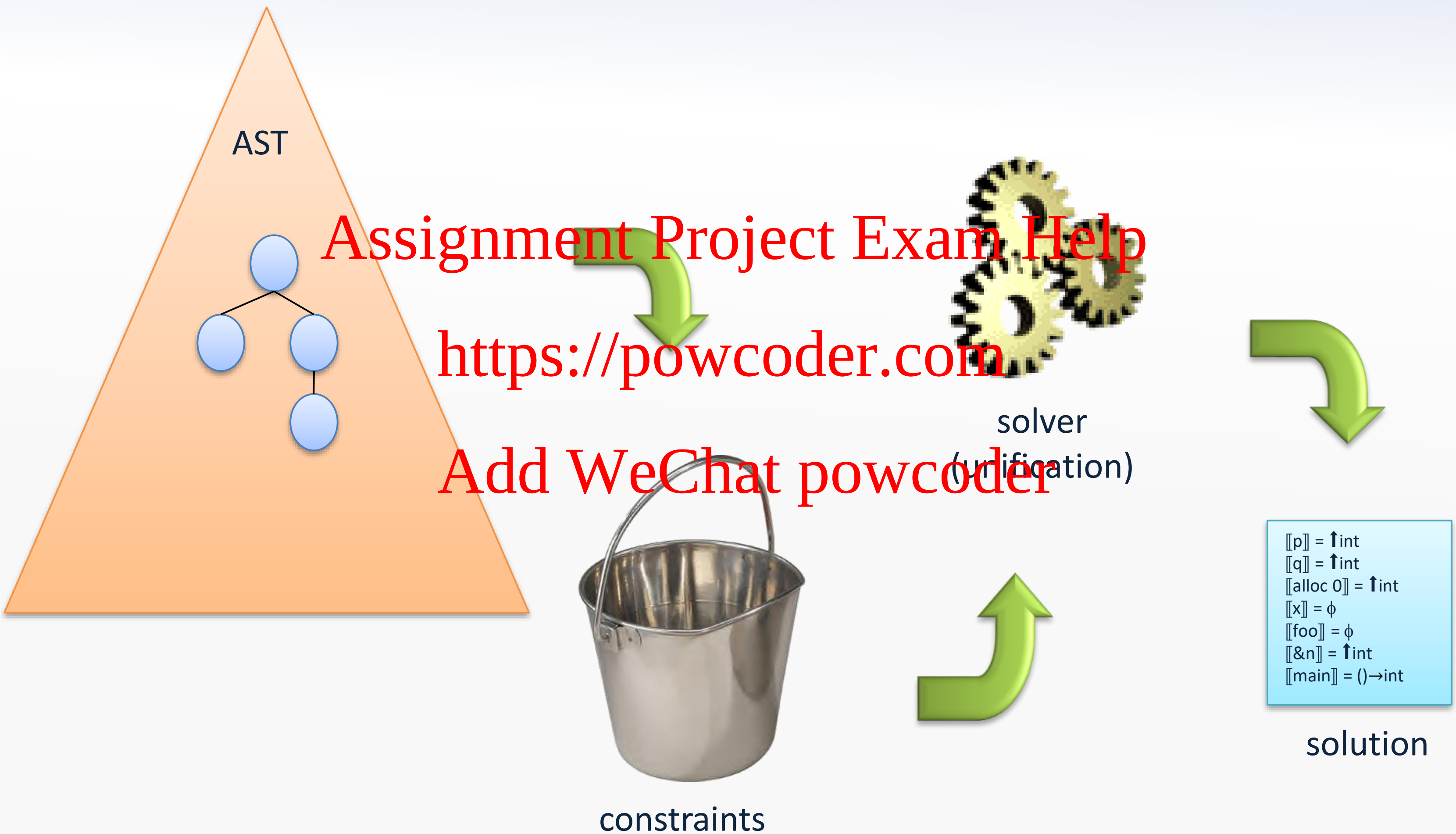
Be careful out there

- The type checker may be unsound:



- Example: covariant arrays in Java
  - a deliberate pragmatic choice

Generating and solving constraints



Types

- Types describe the possible values:

$$
\begin{aligned}
\mathit{Type} \rightarrow\ & \mathtt{int} \\
\mid\ & \uparrow \mathit{Type} \\
\mid\ & (\, \mathit{Type}\, ,\ \ldots\, ,\ \mathit{Type}\,) \rightarrow \mathit{Type} \\
\mid\ & \{\ \mathit{Id} : \mathit{Type}\ ,\ \ldots\ ,\ \mathit{Id} : \mathit{Type}\ \}
\end{aligned}
$$

- These describe integers, pointers, functions, and records

- Types are *terms* generated by this grammar
  - example: $(\mathtt{int}, \uparrow\mathtt{int}) \rightarrow \uparrow\uparrow\mathtt{int}$

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)

I:	$\llbracket I \rrbracket = \texttt{int}$
$E_1\ op\ E_2$:	$\llbracket E_1 \rrbracket = \llbracket E_2 \rrbracket = \llbracket E_1\ op\ E_2 \rrbracket = \texttt{int}$
$E_1 == E_2$:	$\llbracket E_1 \rrbracket = \llbracket E_2 \rrbracket \wedge \llbracket E_1 == E_2 \rrbracket = \texttt{int}$
`input`:	$\llbracket \texttt{input} \rrbracket = \texttt{int}$
$X = E$:	$\llbracket X \rrbracket = \llbracket E \rrbracket$
`output` E:	$\llbracket E \rrbracket = \texttt{int}$
`if` (E) $\{S\}$:	$\llbracket E \rrbracket = \texttt{int}$
`if` (E) $\{S_1\}$ `else` $\{S_2\}$:	$\llbracket E \rrbracket = \texttt{int}$
`while` (E) $\{S\}$:	$\llbracket E \rrbracket = \texttt{int}$

Generating constraints (2/3)

$X(X_1,\ldots,X_n)\{\ \ldots\ \texttt{return}\ E;\ \}$:

$$[\![X]\!] = ([\![X_1]\!], \ldots, [\![X_n]\!]) \rightarrow [\![E]\!]$$

$E(E_1, \ldots, E_n)$:

$$[\![E]\!] = ([\![E_1]\!], \ldots, [\![E_n]\!]) \rightarrow [\![E(E_1, \ldots, E_n)]\!]$$

$\texttt{alloc}\ E$: $\quad [\![\texttt{alloc}\ E]\!] = \uparrow[\![E]\!]$

$\&X$: $\quad [\![\&X]\!] = \uparrow[\![X]\!]$

$\texttt{null}$: $\quad [\![\texttt{null}]\!] = \uparrow\alpha$ $\quad$ (each α is a fresh type variable)

$*E$: $\quad [\![E]\!] = \uparrow[\![*E]\!]$

$*E_1 = E_2$: $\quad [\![E_1]\!] = \uparrow[\![E_2]\!]$

For each parameter X of the main function: $[\![X]\!] = \texttt{int}$

For the return expression E of the main function: $[\![E]\!] = \texttt{int}$

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)

Generating constraints (3/3)

$\{X_1 : E_1, \ldots, X_n : E_n\}$:

$$[\![\{X_1 : E_1, \ldots, X_n : E_n\}]\!] = \{ X_1 : [\![E_1]\!], \ldots, X_n : [\![E_n]\!] \}$$

$E.X$:

$$[\![E]\!] = \{ \ldots, X : [\![E.X]\!], \ldots \}$$

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

Extend $\mathit{Type} \rightarrow \ldots \mid \diamond$ where $\diamond$ represents absent fields

$$\{X_1\!:\!E_1, \ldots, X_n\!:\!E_n\}: \quad [\![\{X_1\!:\!E_1, \ldots, X_n\!:\!E_n\}]\!] = \{f_1\!:\!\gamma_1, \ldots, f_m\!:\!\gamma_m\}$$

$$\text{where } \gamma_i = \begin{cases} [\![E_j]\!] & \text{if } f_i = X_j \text{ for some } j \\ \diamond & \text{otherwise} \end{cases}$$

$$E.X: \quad [\![E]\!] = \{f_1\!:\!\gamma_1, \ldots, f_m\!:\!\gamma_m\} \land [\![E.X]\!] \neq \diamond$$

$$\text{where } \gamma_i = \begin{cases} [\![E.X]\!] & \text{if } f_i = X \\ \alpha_i & \text{otherwise} \end{cases}$$

(Field write statements? Exercise...)

General terms

Constructor symbols:

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

Ex: `int`

Ex: $\&\tau$

Ex: $(\tau_1, \tau_2) \rightarrow \tau_3$

Terms:

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

Terms with variables:

- f(X,b)
- h(X,g(Y,Z))

X, Y, and Z here are type variables, like ⟦`(*p)-1`⟧ or ⟦`p`⟧, not program variables

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 terms that makes both sides equal:

  $X = f(d(b),b)$

  $Y = d(b)$

  $Z = b$

Implicit constraint for term equality:
$c(t_1,\ldots,t_k) = c(t_1',\ldots,t_k') \Rightarrow t_i = t_i'$ 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

$[\![\texttt{null}]\!] = \uparrow t$

$[\![\texttt{alloc null}]\!] = \uparrow [\![\texttt{null}]\!]$

$[\![\texttt{p}]\!] = [\![\texttt{alloc null}]\!]$

$[\![\texttt{p}]\!] = \uparrow [\![\texttt{p}]\!]$

which have this “recursive solution” for p:

$[\![\texttt{p}]\!] = t$ where $t = \uparrow t$

Regular terms

- Infinite but (eventually) repeating:
  - e(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(...),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
}
```

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



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

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 type variables
- 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

⟦foo⟧ = (⟦p⟧,⟦x⟧)→⟦f⟧
⟦*p⟧ = 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)-1⟧ = int

⟦*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⟧ = ⟦0⟧
⟦foo(&n,foo)⟧ = int

Solutions

$[\![p]\!] = \Uparrow\texttt{int}$

$[\![q]\!] = \Uparrow\texttt{int}$

$[\![\texttt{alloc 0}]\!] = \Uparrow\texttt{int}$

$[\![x]\!] = \phi$

$[\![\texttt{foo}]\!] = \phi$

$[\![\&n]\!] = \Uparrow\texttt{int}$

$[\![\texttt{main}]\!] = () \to \texttt{int}$

NO TYPE ERRORS

Here, ϕ is the regular type that is the unfolding of

$$\phi = (\Uparrow\texttt{int}, \phi) \to \texttt{int}$$

which can also be written $\phi = \mu\, t.(\Uparrow\texttt{int},\ t) \to \texttt{int}$

All other variables are assigned $\texttt{int}$

Infinitely many solutions

The function

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


has type $(\uparrow\alpha)\rightarrow\alpha$ for any type α

(which is not expressible in our current type language)

Recursive and polymorphic types

- Extra notation for recursive and polymorphic types:

$$\begin{aligned} \mathit{Type} &\rightarrow \ldots \\ &\mid\ \mu\ \mathit{TypeVar}.\ \mathit{Type} \\ &\mid\ \mathit{TypeVar} \\ \mathit{TypeVar} &\rightarrow t \mid u \mid \ldots \end{aligned}$$

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

- A type $\tau \in \mathit{Type}$ is a (finite) term generated by this grammar
- $\mu\ \alpha.\ \tau$ is the (potentially recursive) type τ where occurrences of α represent τ itself
- $\alpha \in \mathit{TypeVar}$ is a type variable (implicitly universally quantified if not bound by an enclosing μ)

Slack – let-polymorphism

```
f(x) {
  return *x;
}
main() {
  return f(alloc 1) + *(f(alloc(alloc 2));
}
```


This never has a type error at runtime – but it is not typable

$$\uparrow\mathtt{int} = [\![\mathtt{x}]\!] = \uparrow\uparrow\mathtt{int}$$

But we could analyze `f` before `main`: $[\![\mathtt{f}]\!] = (\uparrow\mathtt{t}) \rightarrow \mathtt{t}$

and then "instantiate" that type at each call to `f` in `main`

Slack – let-polymorphism

```
polyrec(g,x) {
  var r;
  if (x==0) {
    r=g;
  } else {
    r=polyrec(2,0);
  }
  return r+1;
}

main() {
   return polyrec(null,1)
}
```


This never has a type error at runtime – but it is not typable

And let-polymorphism doesn't work here because `bar` is recursive

Slack – flow-insensitivity

```
f() {
  var x;
  x = alloc 17;
  x = 42;
  return x + 87;
}
```


This never has a type error at runtime – but it is not typable

The type analysis is *flow insensitive* (it ignores the order of statements)

Other programming errors

- Not all errors are type errors:
  - dereference of `null` pointers
  - reading of uninitialized variables
  - division by zero
  - escaping stack cells

  (why not?)

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

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

- Other kinds of static analysis may catch these