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

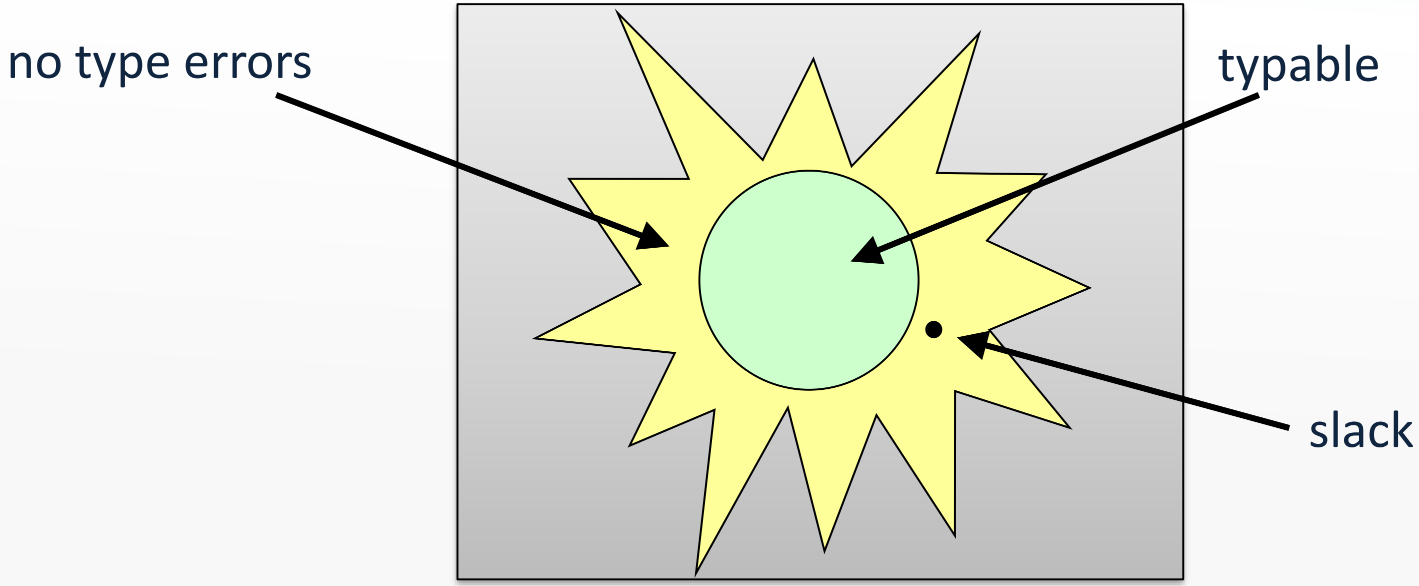
Type checking

- 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*)

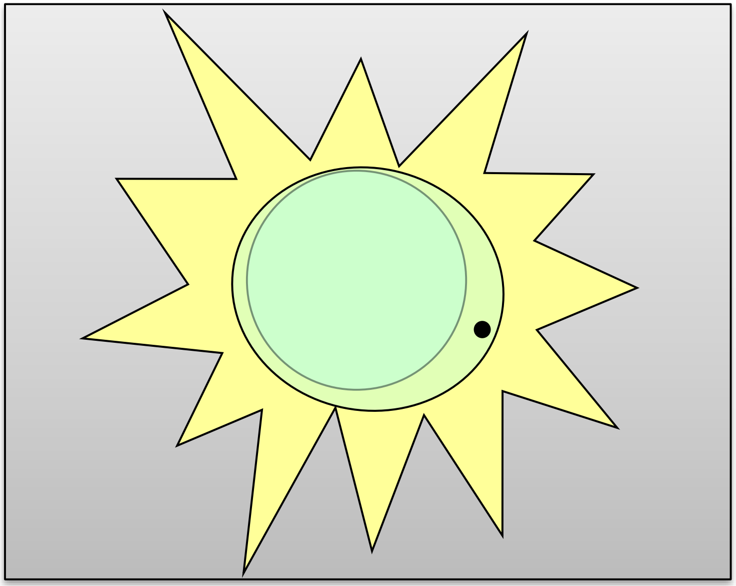
- What we shall see next is the essence of the
  Damas–Hindley–Milner type inference technique,
  which forms the basis of the type systems of e.g. ML, OCaml, and Haskell

Typability



Fighting slack

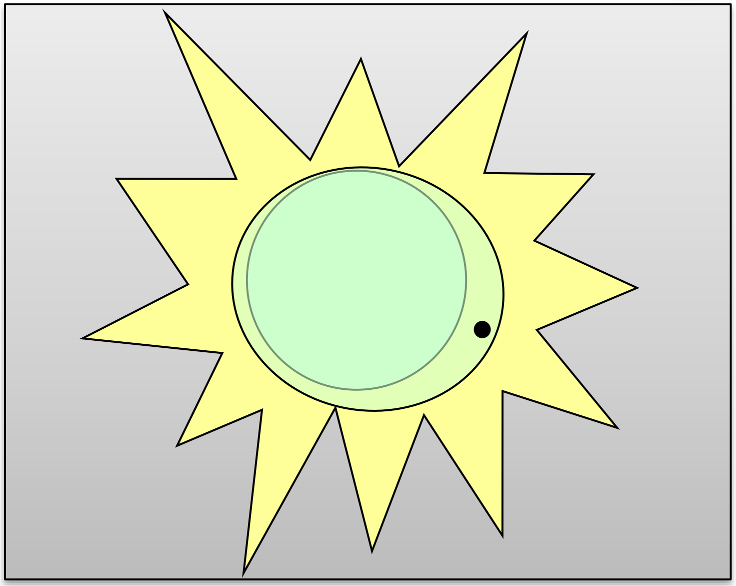
- Make the type checker a bit more clever:



- An eternal struggle

Fighting slack

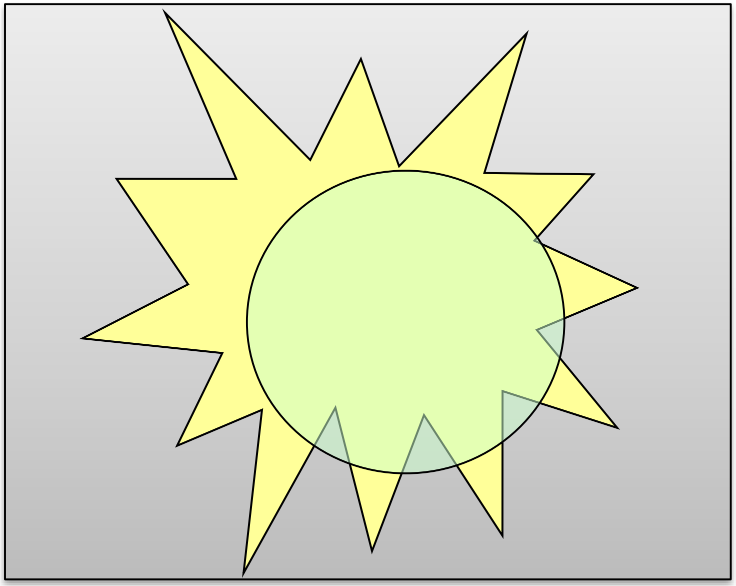
- Make the type checker a bit more clever:



- 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

协变数组

若B是A的子类, 则如下代码在Java中是允许的

A[ ] a=new B[ ];

从类延伸到数组的变换, 原有的继承关系不变

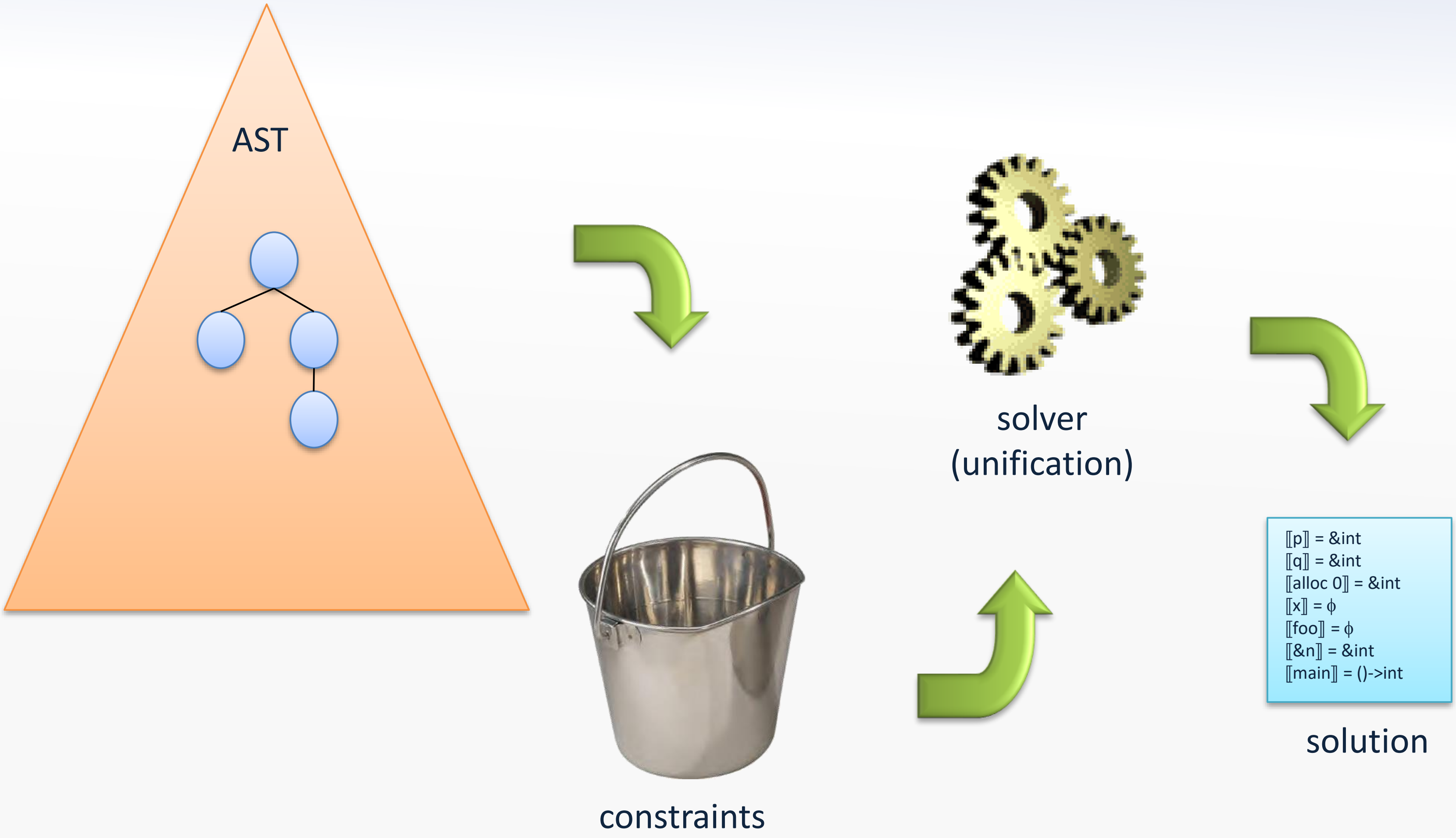
附加：子类型 - 协变和逆变

- □ 子类型关系 <
  - ■ 类型上的偏序关系τ
  - ■ 满足包含原理：如果s是t的子类型，则需要类型为t的值时，都可以将类型为s的值提供给它
- □ 协变（covariant）t<t', 则c(t)<c(t')
  - ■ 函数类型在值域上是协变的

    假设$e:\sigma \rightarrow \tau$，$e1:\sigma$，则$e(e1):\tau$. 如果$\tau < \tau'$，则$e(e1):\tau'$.
- □ 逆变（contravariant）t<t', 则c(t')<c(t)
  - ■ 函数类型在定义域上是逆变的

    假设$e:\sigma \rightarrow \tau$，$e1:\sigma'$，如果$\sigma' < \sigma$，则$e(e1):\tau$.

Generating and solving constraints



AST

constraints

solver
(unification)

⟦p⟧ = &int
⟦q⟧ = &int
⟦alloc 0⟧ = &int
⟦x⟧ = ϕ
⟦foo⟧ = ϕ
⟦&n⟧ = &int
⟦main⟧ = ()->int

solution

Types

- Types describe the possible values:

```
τ → int
  |  &τ
  | ( τ, …, τ ) -> τ
  | { X:τ, …, X:τ }
```

- These describe integers, pointers, functions, and records

- Types are *terms* generated by this grammar
  – example: `(int,&int) -> &&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)

$X(X_1, \ldots, X_n)$`{ … return` E`; }`:	
	$\llbracket X \rrbracket = (\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket) \texttt{->} \llbracket E \rrbracket$
$E(E_1, \ldots, E_n)$:	
	$\llbracket E \rrbracket = (\llbracket E_1 \rrbracket, \ldots, \llbracket E_n \rrbracket) \texttt{->} \llbracket E(E_1, \ldots, E_n) \rrbracket$
`alloc` E:	$\llbracket \texttt{alloc}\ E \rrbracket = \&\llbracket E \rrbracket$
$\&X$:	$\llbracket \&X \rrbracket = \&\llbracket X \rrbracket$
`null`:	$\llbracket \texttt{null} \rrbracket = \&\alpha$ (each α is a fresh type variable)
$*E$:	$\llbracket E \rrbracket = \&\llbracket *E \rrbracket$
$*X = E$:	$\llbracket X \rrbracket = \&\llbracket E \rrbracket$

多态类型

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

E_j

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

$$\text{where } \gamma_i = \begin{cases} [\![E_1]\!] & \text{if } f_i = X_j \text{ for some } j \\ \alpha_i & \text{otherwise} \end{cases}$$

引入类型变量

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

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

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`

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: &τ

Ex: $(\tau_1,\tau_2)\text{->}\tau_3$

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:

闭项：不含类型变量的项

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

$$Y = d(b)$$

$$Z = b$$

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

代换和实例

```
function subst ($t$ : type_exp, Sv: type_var → type_exp ) :
                                                  type_exp;
begin
    if  $t$ 是基本类型  then return $t$
    else if  $t$ 是类型变量 then return $Sv(t)$
    else if  $t$ 是$t_1 \to t_2$ then return
                      $subst(t_1, Sv) \to subst(t_2, Sv)$
end
```

例子（$s < t$ 表示s是t 的实例，α、β是类型变量）

$pointer(\, integer\,) < pointer(\alpha\,)$ $\quad$ $pointer(\, real\,) < pointer(\alpha\,)$

$integer \to integer < \alpha \to \alpha$ $\quad$ $pointer(\alpha\,) < \beta$

$\alpha < \beta$

不合法的实例

例 下面左边的类型表达式不是右边的实例

integer *real*

代换不能用于基本类型

$integer \rightarrow real$ $\alpha \rightarrow \alpha$

α的代换不一致

$integer \rightarrow \alpha$ $\alpha \rightarrow \alpha$

α的所有出现都应该代换

合一

- 合一(unify)
  - 如果存在某个代换 Sv 使得 $S(t_1) = S(t_2)$，那么这两个表达式t_1和 t_2能够合一

- 最一般的合一代换(the most general unifier) S
  - $S(t_1) = S(t_2)$；
  - 对任何其它满足$S'(t_1) = S'(t_2)$的代换Sv'，代换$S'(t_1)$是$S(t_1)$的实例

Recursive data structures

The program

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

creates these constraints

$$\llbracket \texttt{null} \rrbracket = \&\alpha$$
$$\llbracket \texttt{alloc null} \rrbracket = \&\llbracket \texttt{null} \rrbracket$$
$$\llbracket \texttt{p} \rrbracket = \&\llbracket \texttt{alloc null} \rrbracket$$
$$\llbracket \texttt{p} \rrbracket = \&\llbracket \texttt{p} \rrbracket$$

which have this “recursive solution” for p:

$$\llbracket \text{p} \rrbracket = \alpha \quad \text{where } \alpha = \&\alpha$$

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 $n \leq$ Ack(k,k)
  - this is never bigger than 5 for any real value of n

- See the TIP implementation...

https://github.com/cs-au-dk/TIP/blob/master/src/tip/analysis/TypeAnalysis.scala

https://github.com/cs-au-dk/TIP/blob/master/src/tip/solvers/UnionFindSolver.scala

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
}
```

https://github.com/cs-au-dk/TIP/blob/master/src/tip/solvers/UnionFindSolver.scala

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 (plus 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

```
⟦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==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⟧
```

Solutions

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

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

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

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

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

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

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

NO TYPE ERRORS

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

$$\phi = \texttt{(\&int,}\phi\texttt{)->int}$$

which can also be written $\phi = \mu\,\alpha.\texttt{(\&int,}\alpha\texttt{)->int}$

All other variables are assigned `int`

Infinitely many solutions

The function

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

has type (&α)->α for any type α

(which is not expressible in our current type language)

Recursive and polymorphic types

- Extra notation for recursive and polymorphic types:

$$
\begin{aligned}
\tau \rightarrow\ & \dots \\
\mid\ & \mu\, \alpha.\ \tau \\
\mid\ & \alpha
\end{aligned}
$$

(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 τ where occurrences of α represent τ itself
- α is a type variable (implicitly universally quantified if not bound by an enclosing μ)

Slack

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

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

This never causes a type error – but is not typable:

$$\texttt{int} = [\![\texttt{r}]\!] = [\![\texttt{g}]\!] = \&\alpha$$

Other errors

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

  (why not?)

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

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

- Other kinds of static analysis may catch these