Static Program Analysis

Part 1 – the TIP language

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

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

Questions about programs

- Does the program terminate on all inputs?
- How large can the heap become during execution?
- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- Are buffer-overruns possible?
- Data races?
- SQL injections?
- XSS?
- …

Program points

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = alloc 10;
    *q = (*p)-1;
    f=(*p)*(x(q,x));
  }
  return f;
}
```

Invariants:

A property holds at a program point if it holds in any such state for any execution with any input

Questions about program points

- Will the value of `x` be read in the future?
- Can the pointer `p` be `null`?
- Which variables can `p` point to?
- Is the variable `x` initialized before it is read?
- What is a lower and upper bound on the value of the integer variable `x`?
- At which program points could `x` be assigned its current value?
- Do `p` and `q` point to disjoint structures in the heap?
- Can this `assert` statement fail?

Why are the answers interesting?

- Increase efficiency
  - resource usage
  - compiler optimizations
- Ensure correctness
  - verify behavior
  - catch bugs early
- Support program understanding
- Enable refactorings

Testing?

"Program testing can be used to show the presence of bugs, but never to show their absence."

[Dijkstra, 1972]

Nevertheless, testing often takes 50% of the development cost

Programs that reason about programs

a program ***P***

a program analyzer ***A***

P always works correctly

P fails for some inputs

Requirements to the perfect program analyzer

SOUNDNESS (don’t miss any errors)

COMPLETENESS (don’t raise false alarms)

TERMINATION (always give an answer)

Rice's theorem, 1953

CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5](2), and with ideas which are well summarized in the first sections of a paper of Post [7].

I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumer-

COROLLARY B. *There are no nontrivial c.r. classes by the strong definition.*

Rice's theorem

Any non-trivial property of the behavior of programs in a Turing-complete language is undecidable!

Reduction to the halting problem

- Can we decide if a variable has a constant value?

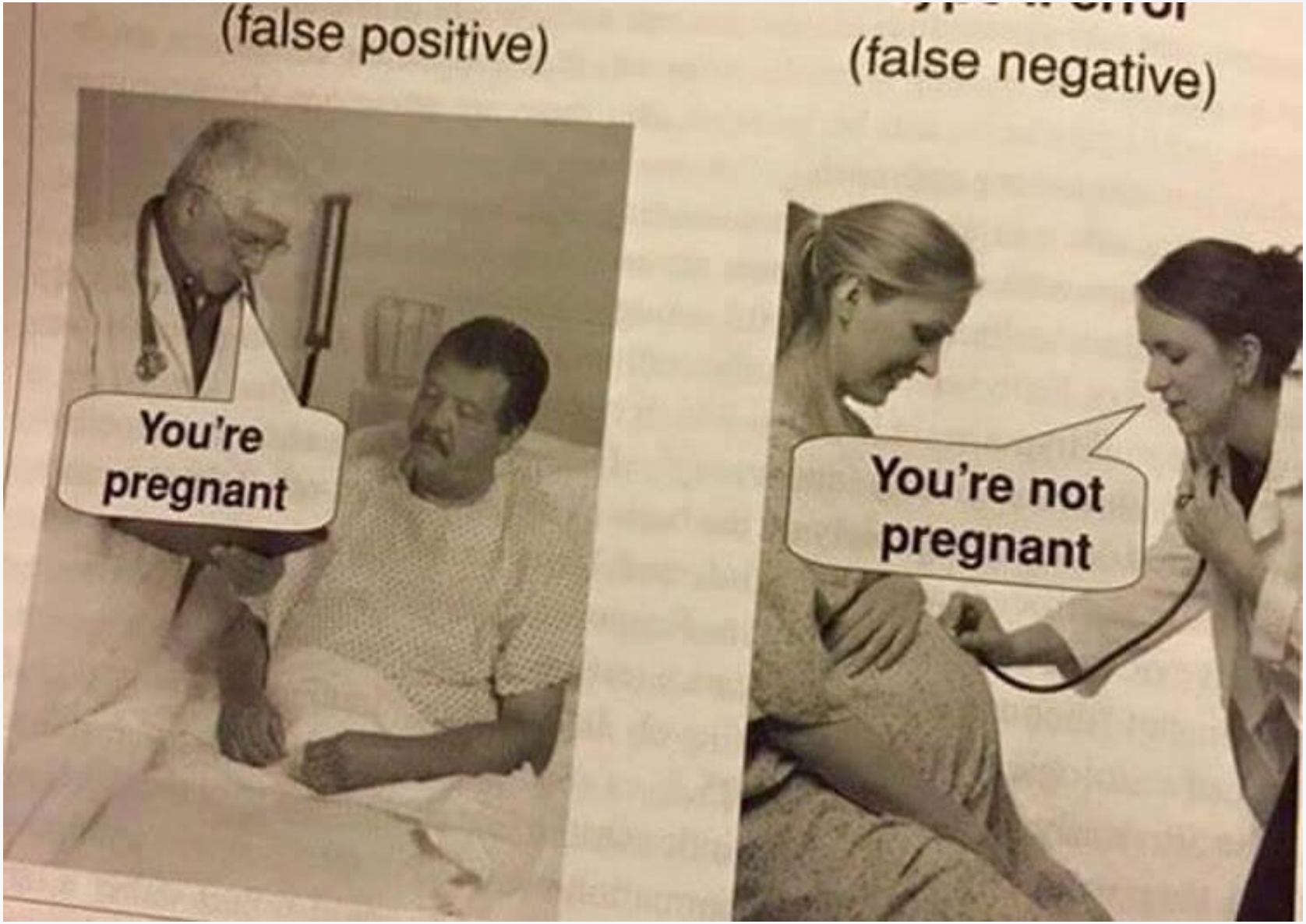
```
x = 17; if (TM(j)) x = 18;
```

- Here, x is constant if and only if the *j*'th Turing machine does not halt on empty input

Approximation

- *Approximate* answers may be decidable!
- The approximation must be *conservative*:
  - i.e. only err on “the safe side”
  - which direction depends on the *client application*
- We'll focus on decision problems
- More subtle approximations if not only “*yes*”/“*no*”
  - e.g. memory usage, pointer targets

False positives and false negatives



Example approximations

- Decide if a given function is ever called at runtime:
  - if "*no*", remove the function from the code
  - if "*yes*", don't do anything
  - the "*no*" answer *must* always be correct if given
- Decide if a cast `(A)x` will always succeed:
  - if "*yes*", don't generate a runtime check
  - if "*no*", generate code for the cast
  - the "*yes*" answer *must* always be correct if given

Beyond “yes”/“no” problems

- How much memory / time may be used in any execution?

- Which variables may be the targets of a pointer variable p?

The engineering challenge

- A correct but trivial approximation algorithm may just give the useless answer every time
- The *engineering challenge* is to give the useful answer often enough to fuel the client application
- ... and to do so within reasonable time and space
- This is the hard (and fun) part of static analysis!

Bug finding

```
int main() {
  char *p,*q;
  p = NULL;
  printf("%s",p);
  q = (char *)malloc(100);
  p = q;
  free(q);
  *p = 'x';
  free(p);
  p = (char *)malloc(100);
  p = (char *)malloc(100);
  q = p;
  strcat(p,q);
}
```

```
gcc -Wall foo.c
lint foo.c

No errors!
```

https://en.wikipedia.org/wiki/Lint_(software)

Does a

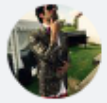
software devel

196 6 68

Uber Engineering

POSTED ON SEP 6, 2017 TO ANDROID, DEVELOPER TOOLS, IO

Finding inter-pr
Infer static anal


SAM BLACKSHEAR

DIN

The capabilities of static analyzers, w
our work on the Infer static analyzer,
source analysis tools like Findbugs,
procedural bugs, or bugs that involve

We'll take a look at two examples of
source DuckDuckGo Android app, ar
the tools mentioned above, which pe
Analyzer — only intra-file analysis (p
unit, a file-with-includes).

Inter-procedural bugs are significant,
Facebook developers have fixed tho
can have a large impact; we include
Facebook. As we have found, inter-p
codebases that consist of millions of

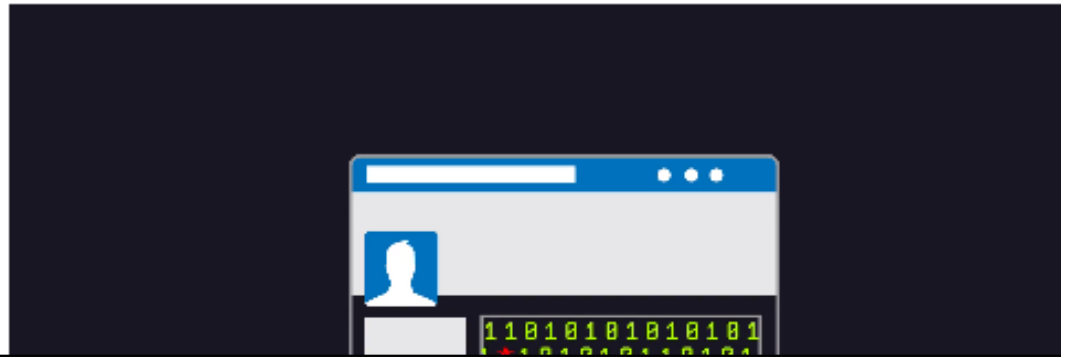
WIRED How Facebook Catches Bugs in Its 100 Million

BUSINESS CULTURE GEAR IDEAS

SHARE

LILY HAY NEWMAN SECURITY 08.15.19 05:03 PM

HOW FACEBOOK CATCHES BUGS IN ITS 100 MILLION LINES OF CODE



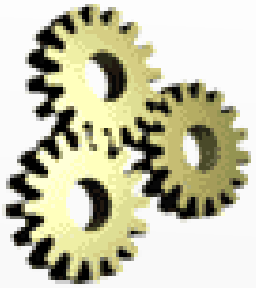
A constraint-based approach

Conceptually separates the analysis specification from algorithmic aspects and implementation details

```
public class Matrix {
public static void main(String[] args) {
        int arr[][]=new int[3][3];
        System.out.println("Enter nine elements");
        Scanner sc=new Scanner(System.in);
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<arr.length;j++)
            {
                arr[i][j]=sc.nextInt();
            }
        }
        int sum=0;
        for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
        if (i == j)
        sum = sum + arr[i][j];
        }}
       System.out.println(sum);
    }
}
```

program to analyze

mathematical constraints



constraint solver

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

solution

Challenging features in modern programming language

- Higher-order functions
- Mutable records or objects, arrays
- Integer or floating-point computations
- Dynamic dispatching
- Inheritance
- Exceptions
- Reflection
- …

The TIP language

- ***T***iny ***I***mperative ***P***rogramming language
- Example language used in this course:
  - minimal C-style syntax
  - cut down as much as possible
  - enough features to make static analysis challenging and fun
- Scala implementation available

Expressions

$$
\begin{aligned}
Exp \rightarrow\ & Int \\
\mid\ & Id \\
\mid\ & Exp + Exp \mid Exp - Exp \mid Exp * Exp \mid Exp \,/\, Exp \\
\mid\ & Exp > Exp \mid Exp == Exp \\
\mid\ & (\ Exp\ ) \\
\mid\ & \texttt{input}
\end{aligned}
$$

- $I \in Int$ represents an integer literal
- $X \in Id$ represents an identifier (x, y, z,…)
- `input` expression reads an integer from the input stream
- comparison operators yield 0 (false) or 1 (true)

Statements

Stm → *Id* = *Exp* ;
| `output` *Exp* ;
| *Stm Stm*
|
| `if (`*Exp*`) {` *Stm* `}` [`else {` *Stm* `}`]?
| `while (`*Exp*`) {` *Stm* `}`

- In conditions, 0 is false, all other values are true
- The `output` statement writes an integer value to the output stream

Functions

- Functions take any number of arguments and return a single value:

```
Fun → Id ( Id , … , Id ) {
        [var Id , … , Id ;]?
        Stm
        return Exp ;
      }
```

- The optional `var` block declares a collection of uninitialized variables
- Function calls are an extra kind of expressions:

```
Exp → … | Id ( Exp , … , Exp )
```

Pointers

Exp → ...
| `alloc` *Exp*
| `&` *Id*
| `*` *Exp*
| `null`

Stm → ... | `*`*Exp* = *Exp* `;`

(No pointer arithmetic)

Records

$$
\begin{aligned}
Exp \rightarrow\ & \ldots \\
& \mid \{\ Id : Exp\ ,\ \ldots\ ,\ Id : Exp\ \} \\
& \mid\ Exp\ .\ Id
\end{aligned}
$$

$$
\begin{aligned}
Stm \rightarrow\ & \ldots \\
& \mid\ Id\ .\ Id = Exp\ ; \\
& \mid\ (*Exp)\ .\ Id = Exp\ ;
\end{aligned}
$$

Records are passed by value (like structs in C)

For simplicity, values of record fields cannot themselves be records

Functions as values

- Functions are first-class values
- The name of a function is like a variable that refers to that function

- Generalized function calls:

$$Exp \rightarrow \ldots \mid Exp\,(\,Exp\,,\,\ldots\,,\,Exp\,)$$

- Function values suffice to illustrate the main challenges with methods (in object-oriented languages) and higher-order functions (in functional languages)

Programs

- A program is a collection of functions
- The function named `main` initiates execution
  - its arguments are taken from the input stream
  - its result is placed on the output stream
- We assume that all declared identifiers are unique

Prog → *Fun ... Fun*

An iterative factorial function

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

A recursive factorial function

```
rec(n) {
  var f;
  if (n==0) {
    f=1;
  } else {
    f=n*rec(n-1);
  }
  return f;
}
```

An unnecessarily 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);
}
```

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

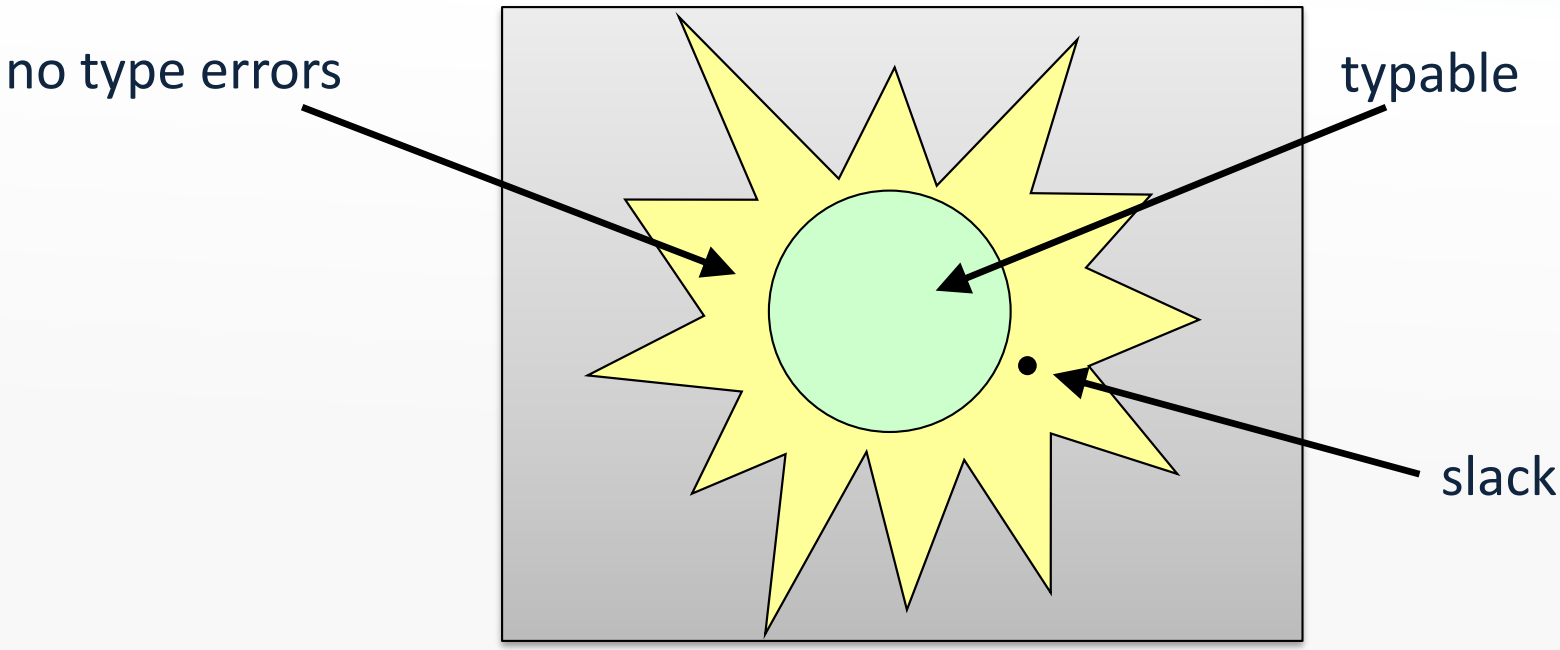
Type checking

- 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*)

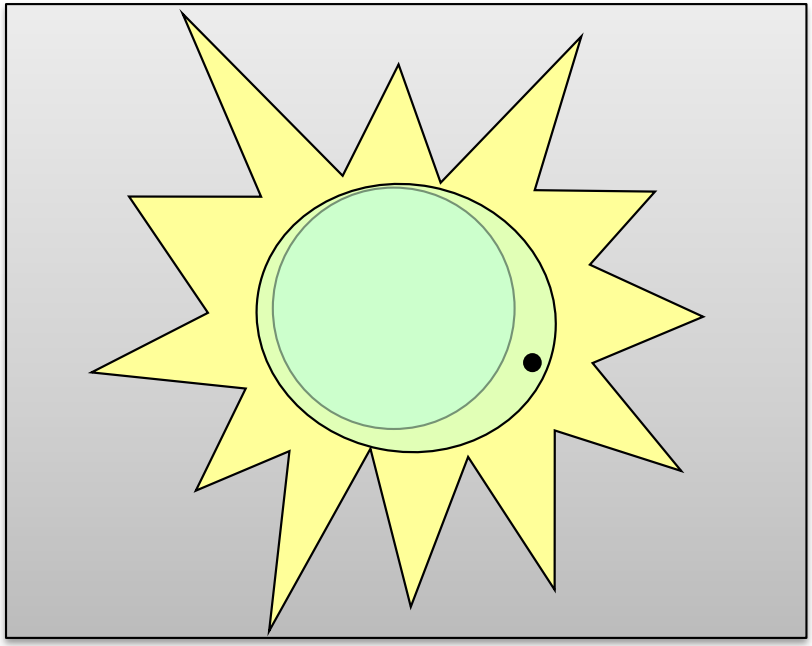
- What we shall see next is the essence of the Damas–Hindley–Milner type inference technique, which forms the basis of the type systems of e.g. ML, OCaml, and Haskell

Typability



Fighting slack

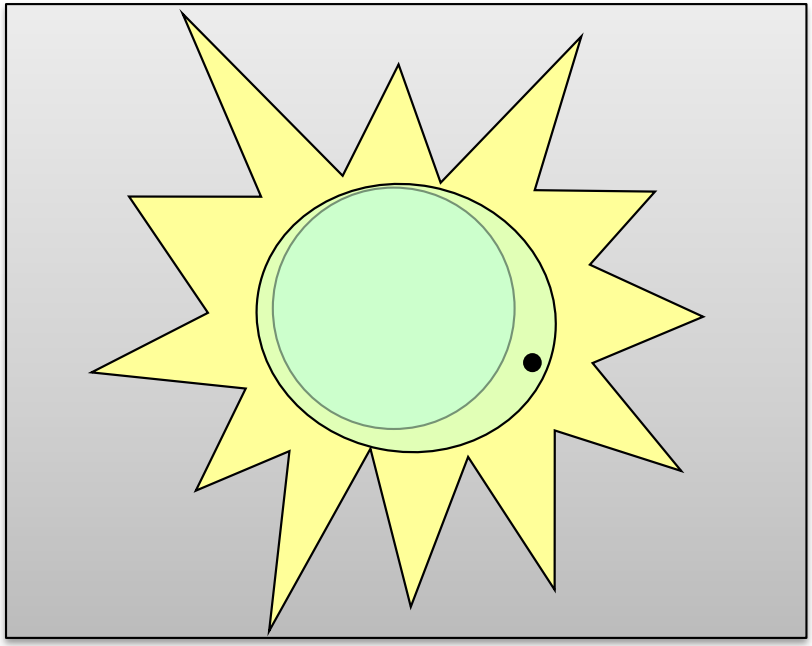
- Make the type checker a bit more clever:



- An eternal struggle

Fighting slack

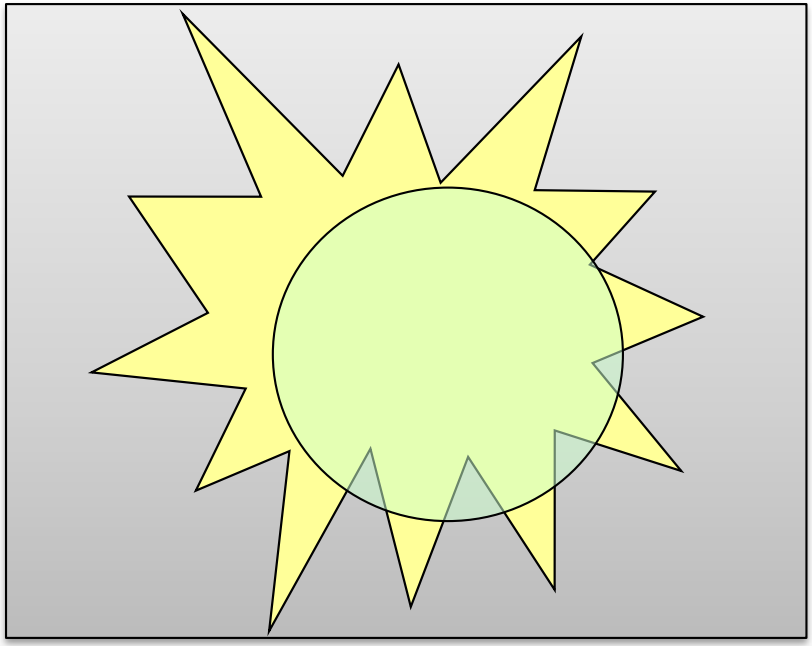
- Make the type checker a bit more clever:



- An eternal struggle
- And a great source of publications

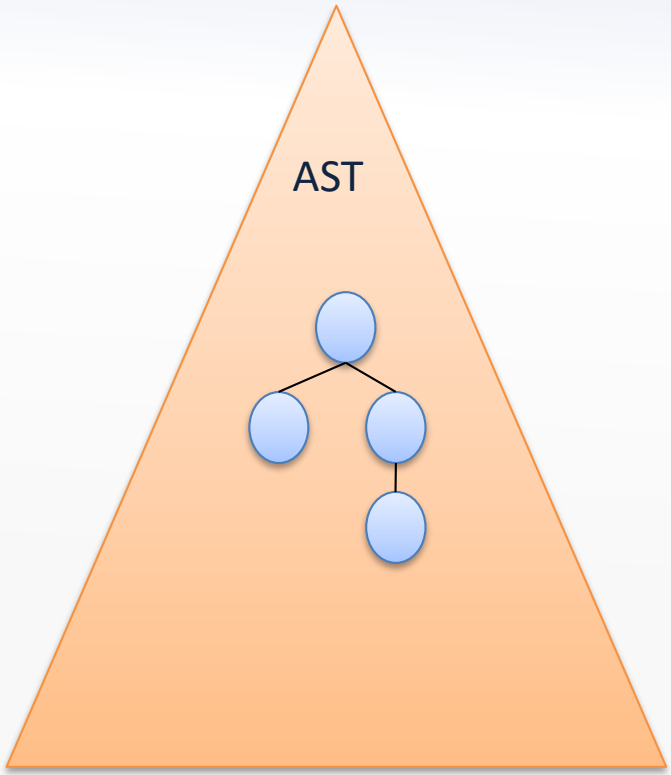
Be careful out there

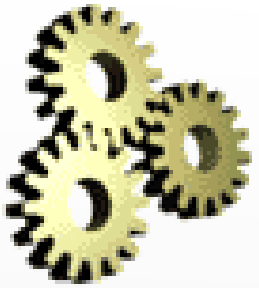
- The type checker may be unsound:



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

Generating and solving constraints





solver
(unification)

constraints

[[p]] = ↑int
[[q]] = ↑int
[[alloc 0]] = ↑int
[[x]] = ϕ
[[foo]] = ϕ
[[&n]] = ↑int
[[main]] = ()→int

solution

Types

- Types describe the possible values:

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

- These describe integers, pointers, functions, and records

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

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:	$[\![I]\!] = \texttt{int}$
E_1 *op* E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \textit{ op } E_2]\!] = \texttt{int}$
$E_1 == E_2$:	$[\![E_1]\!] = [\![E_2]\!] \wedge [\![E_1 == E_2]\!] = \texttt{int}$
`input`:	$[\![\texttt{input}]\!] = \texttt{int}$
$X = E$:	$[\![X]\!] = [\![E]\!]$
`output` E:	$[\![E]\!] = \texttt{int}$
`if (`E`) {`S`}`:	$[\![E]\!] = \texttt{int}$
`if (`E`) {`S_1`} else {`S_2`}`:	$[\![E]\!] = \texttt{int}$
`while (`E`) {`S`}`:	$[\![E]\!] = \texttt{int}$

Generating constraints (2/3)

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

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

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

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

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

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

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

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

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

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 $Type \rightarrow \ldots \mid \diamond$ where $\diamond$ represents absent fields

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

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

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

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

(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: `&τ`

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

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:

$[\![p]\!] = t$ 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...

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

$[\![p]\!] = \uparrow\texttt{int}$
$[\![q]\!] = \uparrow\texttt{int}$
$[\![\texttt{alloc 0}]\!] = \uparrow\texttt{int}$
$[\![x]\!] = \phi$
$[\![\texttt{foo}]\!] = \phi$
$[\![\&n]\!] = \uparrow\texttt{int}$
$[\![\texttt{main}]\!] = \texttt{()}\rightarrow\texttt{int}$

NO TYPE ERRORS

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

$$\phi = (\uparrow\texttt{int}, \phi)\rightarrow\texttt{int}$$

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

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

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

But we could analyze `f` before `main`: $[\![f]\!] = (\uparrow t) \to t$

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