Introduction
A Tiny Example Language
Type Analysis

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

Questions About Programs

- Does the program terminate?
- How large can the heap become during execution?
- What is the possible output?

Program Points

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

any point in the program
= any value of the PC

- 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?

Why Are The Answers Interesting?

- Ensure correctness:
  - verify behavior
  - catch bugs early
- Increase efficiency:
  - resource usage
  - compiler optimizations

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!

Easy Reduction

- 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 halts on empty input.

Approximation

- *Approximate* answers may be decidable!
- The approximation must be *conservative*:
  - either ”*yes*” or ”*no*” must always be the correct answer
  - which direction depends on the *client application*
  - the useful answer must always be correct
- More subtle approximations if not only ”*yes*”/”*no*”

Example Approximations

- Decide if a 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

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

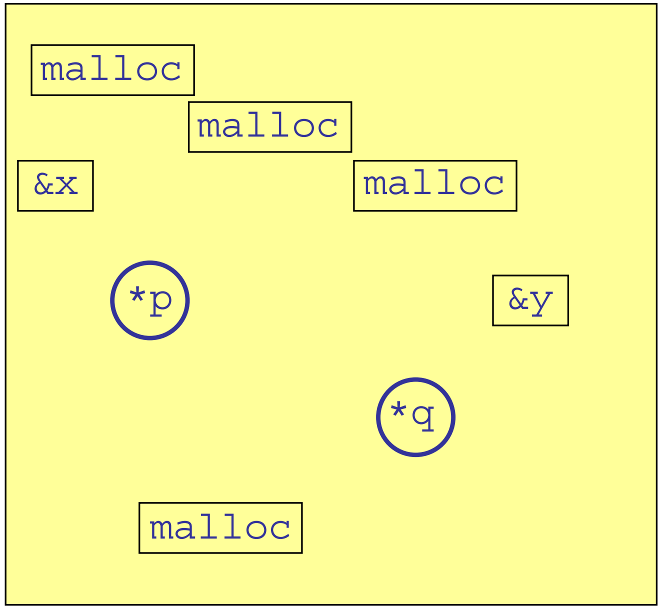
- If we want to replace `*p` by `x`:
  - answer `&x` if `x` is guaranteed to be the only target
  - answer ? otherwise

- If we want to know the maximal size of `*p`:
  - answer {`&x`, `&y`, `&z`, ...}
  - guaranteed to contain all targets (but may be too large)

The Engineering Challenge

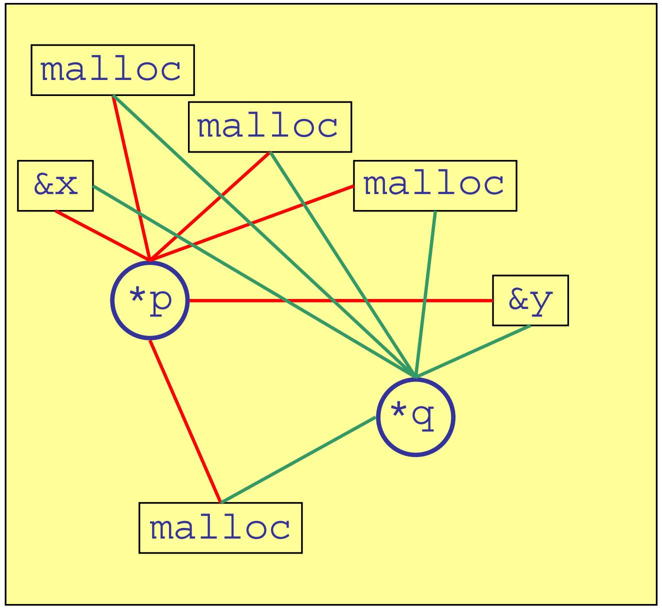
- A correct but trivial approximation algorithm may just give the useless answer every time
- The *engineering challenge* is to give the useful answer often enough to fuel the client application
- This is the hard (and fun) part of static analysis...

Engineering in Practice (1/4)



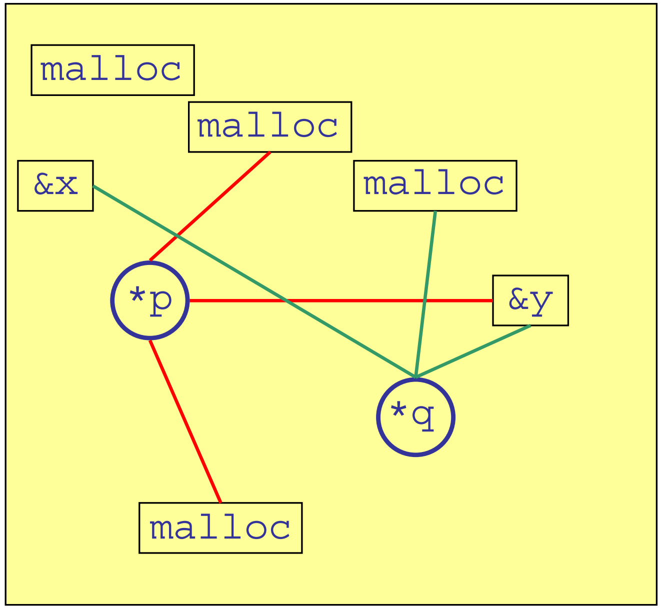
- Where do the pointers come from?

Engineering in Practice (2/4)



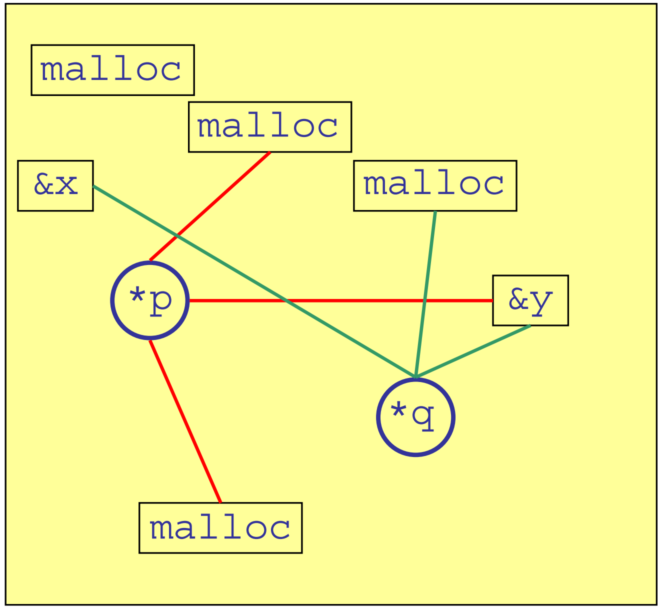
- The trivial answer: from somewhere!

Engineering in Practice (3/4)



- The hard answer: from a few places!

Engineering in Practice (4/4)



- Over the last 15 years:

  ≥ 500 publications, ≥ 50 PhD theses

The Phases of GCC (1/2)

Parsing
Tree optimization
RTL generation
Sibling call optimization
Jump optimization
Register scan
Jump threading
Common subexpression elimination
Loop optimizations
Jump bypassing
Data flow analysis
Instruction combination
If-conversion
Register movement
Instruction scheduling
Register allocation
Basic block reordering
Delayed branch scheduling
Branch shortening
Assembly output
Debugging output

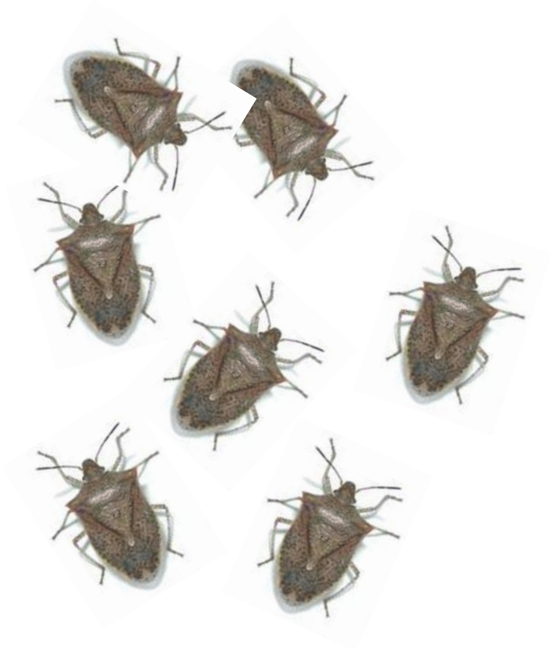
The Phases of GCC (2/2)

- Parsing
- Tree optimization
- RTL generation
- Sibling call optimization
- Jump optimization
- Register scan
- Jump threading
- Common subexpression elimination
- Loop optimizations
- Jump bypassing
- Data flow analysis
- Instruction combination
- If-conversion
- Register movement
- Instruction scheduling
- Register allocation
- Basic block reordering
- Delayed branch scheduling
- Branch shortening
- Assembly output
- Debugging output

Static analysis uses 60% of the compilation time

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!
```

Optimizations or Bug Finding?

- Moore's Law:

  Advances in hardware doubles computing power every 18 months

- Proebsting's Law

  Advances in optimization techniques doubles computing power every 18 years

- So why bother with compiler optimizations?

An Eternal Struggle

- Compiler optimizations yield a nearly constant speedup factor of 4
- Nobody would give that up regardless of processor speed!
- Increases in language abstractions require constant progress just to stay in place

The Optimizer Must Undo Abstractions

- Variables abstract away from register
  - the optimizer must find an efficient mapping
- Control structures abstract away from gotos
  - the optimizer must find and simplify the goto graph
- Data structures abstract away from memory
  - the optimizer must find an efficient layout

...

- Methods abstract away from procedures
  - the optimizer must find the intended implementation

The Case of BETA

- The BETA language unifies as *patterns*:
  - abstract classes
  - concrete classes
  - methods
  - functions

- A (hypothetical) optimizing BETA compiler must classify patterns to recover this information
- Example: all patterns are heap-allocated, but 50% are methods that could be stack-allocated

Static Analysis Concepts

- Constraints and solutions
- Lattices, fixed-points, equations
- Flow-sensitive vs. flow-insensitive
- Control flow graph
- Dataflow analysis
- Widening and narrowing
- Interprocedural vs. intraprocedural
- Polyvariance vs. monovariance
- Control flow analysis
- Pointer analysis
- Shape analysis
- Context- and path-sensitivity

this course

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
  - all concepts that make static analysis challenging

- Java implementation available
  - thanks to Johnni Winther

Expressions

$$
\begin{aligned}
E \quad & \rightarrow \mathit{intconst} \\
& \rightarrow \mathit{id} \\
& \rightarrow E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid E{>}E \mid E{==}E \\
& \rightarrow (\ E\ ) \\
& \rightarrow \mathtt{input}
\end{aligned}
$$

- The `input` expression reads an integer from the input stream
- The comparison operators yield 0 and 1

Statements

$$
\begin{aligned}
S \;&\rightarrow id = E\texttt{;}\\
&\rightarrow \texttt{output}\ E\texttt{;}\\
&\rightarrow S\ S\\
&\rightarrow \texttt{if (}E\texttt{) \{} S \texttt{\}}\\
&\rightarrow \texttt{if (}E\texttt{) \{} S \texttt{\} else \{} S \texttt{\}}\\
&\rightarrow \texttt{while (}E\texttt{) \{} S \texttt{\}}\\
&\rightarrow \texttt{var}\ id, \ldots, id\texttt{;}
\end{aligned}
$$

Statements

- In conditions, 0 is false, all other values are true
- The `output` statement writes an integer value to the output stream
- The `var` statement declares a collection of uninitialized variables

Functions

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

$$F \rightarrow id \ (\ id \ , \ \ldots \ , \ id \ ) \\ \{ \ \texttt{var} \ id \ , \ \ldots \ , \ id \ ; \ S \ \texttt{return} \ E \ \}$$

- Function calls are an extra kind of expressions:

$$E \rightarrow id \ (\ E \ , \ \ldots \ , \ E \ )$$

Dynamic Memory

$$
\begin{aligned}
E &\to \&id \\
&\to \texttt{malloc} \\
&\to *E \\
&\to \texttt{null} \\
\\
S &\to *id = E\,;
\end{aligned}
$$

- Pointer arithmetics is not permitted

Function Pointers

- Function names denote function pointers
- Generalized function calls:

$$E \rightarrow (E)\,(\,E\,,\ \ldots,\ E\,)$$

- Function pointers are a simple model for objects or higher-order functions

Programs

- A program is a collection of functions
- The final function 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

$$P \quad \rightarrow F \ldots F$$

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 = malloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

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

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

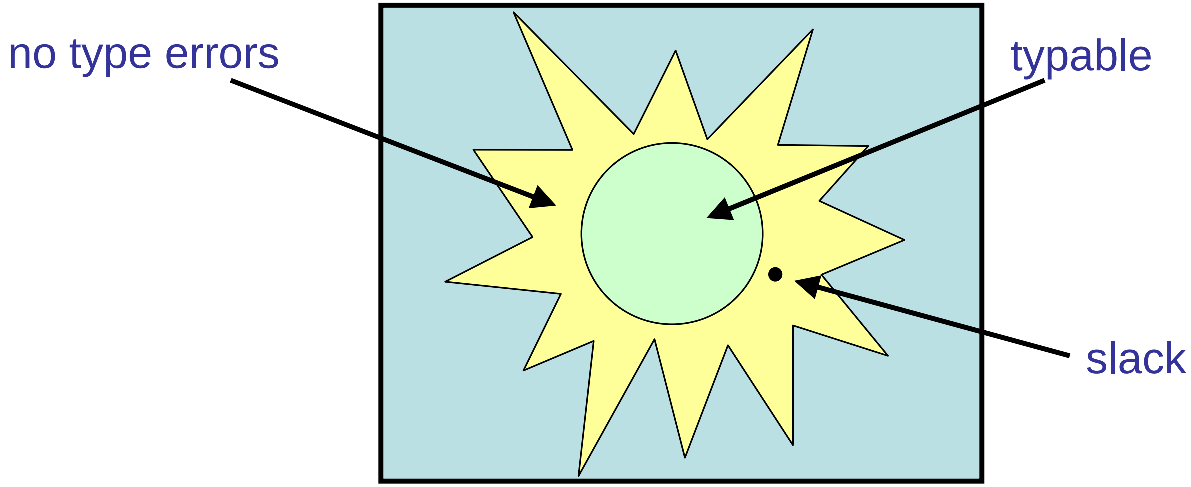
- Violations result in runtime errors

Type Checking

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

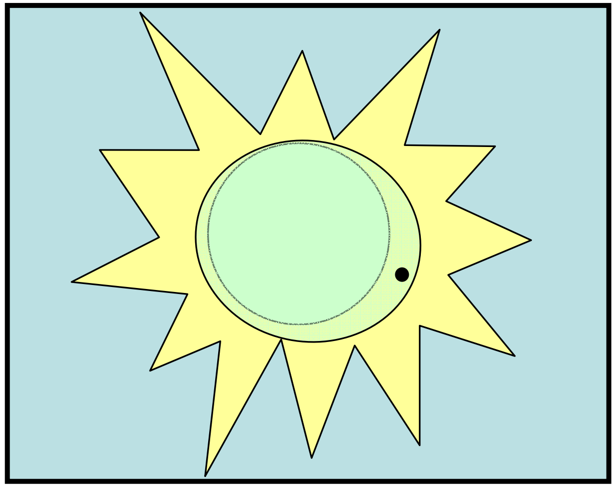
- Instead, we use conservative approximation
  - a program is *typable* if it satisfies some *type constraints*
  - these are systematically derived from the syntax tree
  - if typable, then no runtime errors occur
  - but some programs will be unfairly rejected (*slack*)

Typability



Fighting Slack

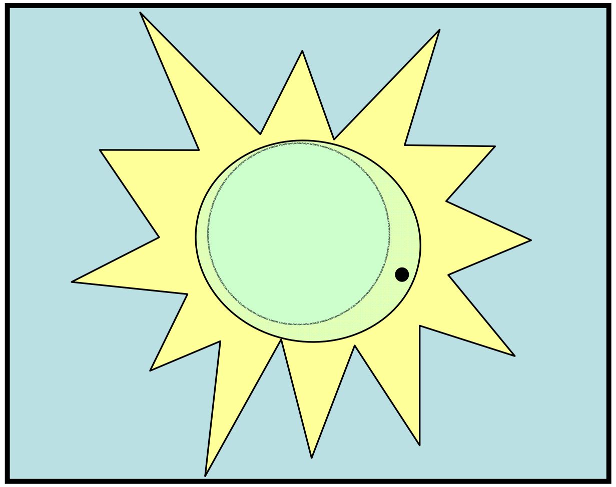
- Make the type checker a bit more clever:



- An eternal struggle

Fighting Slack

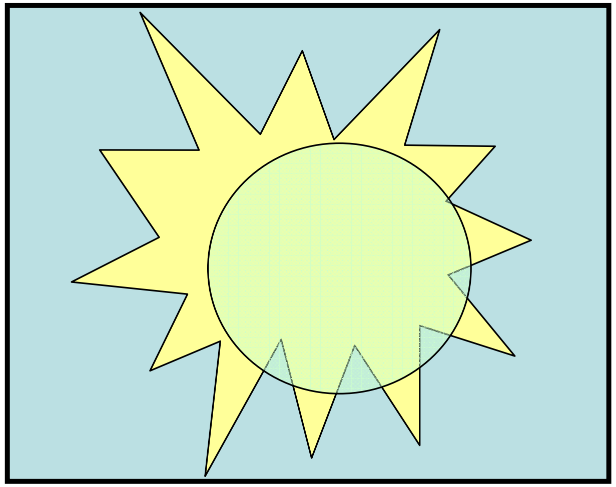
- Make the type checker a bit more clever:



- 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

Types

- Types describe the possible values:

$$
\begin{aligned}
\tau \;&\rightarrow \texttt{int} \\
&\rightarrow \texttt{\&}\tau \\
&\rightarrow \texttt{(} \tau\texttt{,} \ldots\texttt{,} \tau \texttt{)} \;\texttt{->}\; \tau
\end{aligned}
$$

- These describe integers, pointer, and functions
- Types are *terms* generated by this grammar

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 terms that makes both sides equal:

X = f(d(b),b)

Y = d(b)

Z = b

Unification Errors

- Constructor error:

```
d(X) = e(X)
```

- Arity error:

```
a = a(X)
```

The Unification Algorithm

- Paterson and Wegman (1976)
- In time O(*n*):
  - finds a most general unifier
  - or decides that none exists
- This is used as a backend for type checking

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

- Paterson and Wegman (1976)
- The unification problem can be solved in $O(n\alpha(n))$

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

Type Constraints

- We generate type constraints from an AST:
  - all constraints are equalities
  - they can be solved using the unification algorithm
- Type variables:
  - for each identifier *id* we have the variable [[*id*]]
  - for each expression *E* we have the variable [[*E*]]
- Recall that all identifiers are unique
- The expression *E* is an AST node, not syntax

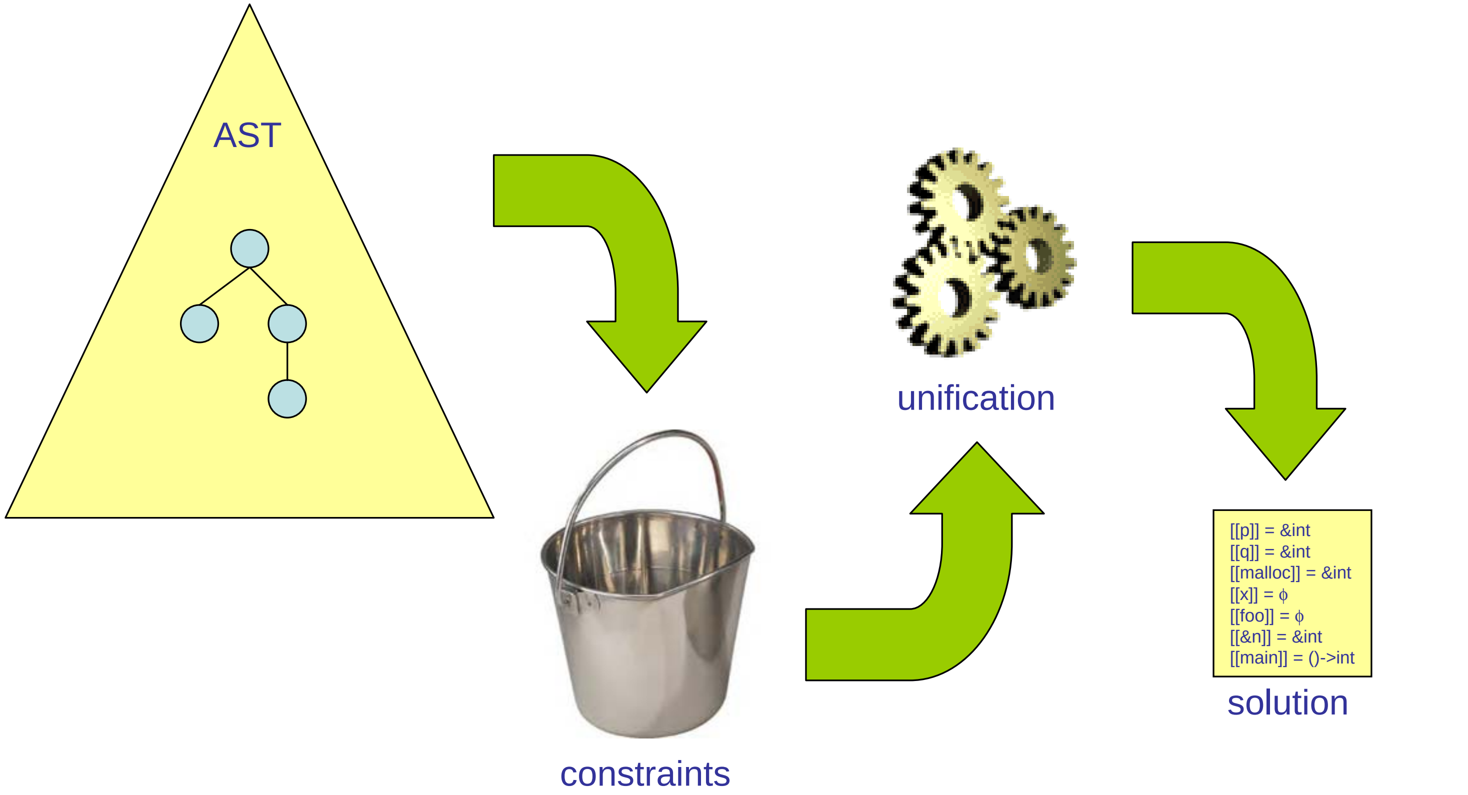
Generating Constraints (1/2)

intconst:	$[[intconst]]$ = `int`
E_1 *op* E_2:	$[[E_1]] = [[E_2]] = [[E_1 \; op \; E_2]]$ = `int`
E_1 `==` E_2:	$[[E_1]] = [[E_2]] \wedge [[E_1{==}E_2]]$ = `int`
`input`:	[[`input`]] = `int`
id `=` *E*:	$[[id]] = [[E]]$
`output` *E*:	$[[E]]$ = `int`
`if (`*E*`) {`*S*`}`:	$[[E]]$ = `int`
`if (`*E*`) {`S_1`} else {`S_2`}`:	$[[E]]$ = `int`
`while (`*E*`) {`*S*`}`:	$[[E]]$ = `int`

Generating Constraints (2/2)

id`(`id_1`, ...,`id_n`){ ... return `E`; }:`	
	$[[id]]$ = `(`$[[id_1]]$, ..., $[[id_n]]$`) ->` $[[E]]$
`(`E`)(`E_1`, ...,` E_n`):`	
	$[[E]]$ = ($[[E_1]]$, ..., $[[E_n]]$) `->` [[`(`E`)(`E_1`, ...,` E_n`)`]]
`&`id:	[[`&`id]] = `&`$[[id]]$
`malloc`:	[[`malloc`]] = `&`α
`null`:	[[`null`]] = `&`α
`*`E:	$[[E]]$ = `&`[[`*`E]]
`*`id `=` E:	$[[id]]$ = `&`$[[E]]$

Generating and Solving Constraints



The Complicated Function

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = malloc;
    *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]]
[[malloc]] = &α
[[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]] = [[malloc]]
[[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

Solutions

[[p]] = `&int`

[[q]] = `&int`

[[malloc]] = `&int`

[[x]] = ϕ

[[foo]] = ϕ

[[&n]] = `&int`

[[main]] = `()->int`

NO TYPE ERRORS

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

  ϕ = `(&int,`ϕ`)->int`

- All other variables are assigned `int`

Recursive Data Structures

- The program:

```
var p;
p = malloc;
*p = p;
```

creates the constraints:

$[[p]] = \&\alpha$

$[[p]] = \&[[p]]$

which have the solution:

$[[p]] = \psi$, where $\psi = \&\psi$

Infinitely Many Solutions

- The function:

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

has type $\&\alpha \text{->} \alpha$ for any type α

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 cause a type error, but is not typable:
  `int` = [[`r`]] = [[`g`]] = `&`α

Other Errors

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

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

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

- Other kinds of static analysis may catch these

Flow-Sensitivity

- Type checking is flow-*insensitive*:
  - statements may be permuted without affecting typability
  - constraints are generated from *AST nodes*

- Other analyses must be flow-*sensitive*:
  - the flow of statements affects the results
  - constraints are generated from *flow graph nodes*