Semantic and Context Analysis

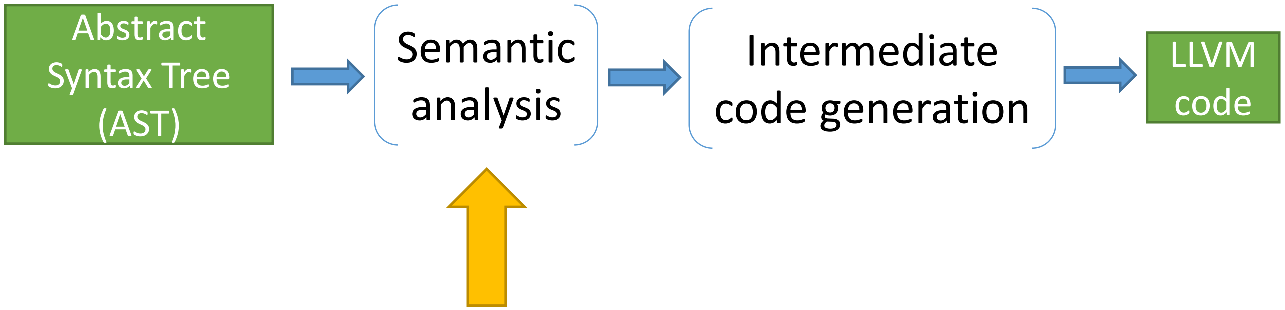
Yotam Feldman
Guy Gueta
Mooly Sagiv

Motivation

Silly Java Program

```
                             Interface not declared
class MyClass implements MyInterface {
string myInteger;
void doSomething() {
  int[] x = new string;      Type mismatch
  x[5] = myInteger * y   ;   y is undefined
  }        Can't multiply Strings
void doSomething() {  Can't redefine functions
  }
int fibonacci(int n) {
   return doSomething() + fibonacci(n – 1);
  }          Can't add void
}
```

Semantic Analysis



Abstract Syntax Tree (AST)

Semantic analysis

Intermediate code generation

LLVM code

Semantic Analysis

Syntactically valid programs may be erroneous

```
int a;
a = “hello”;
```

Assigning wrong type

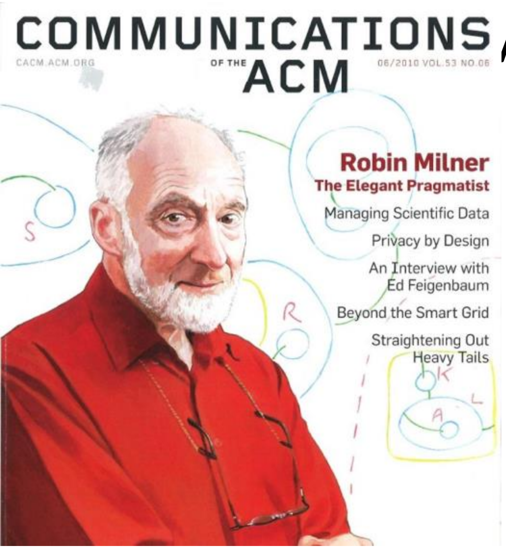
```
int a;
b = 1;
```

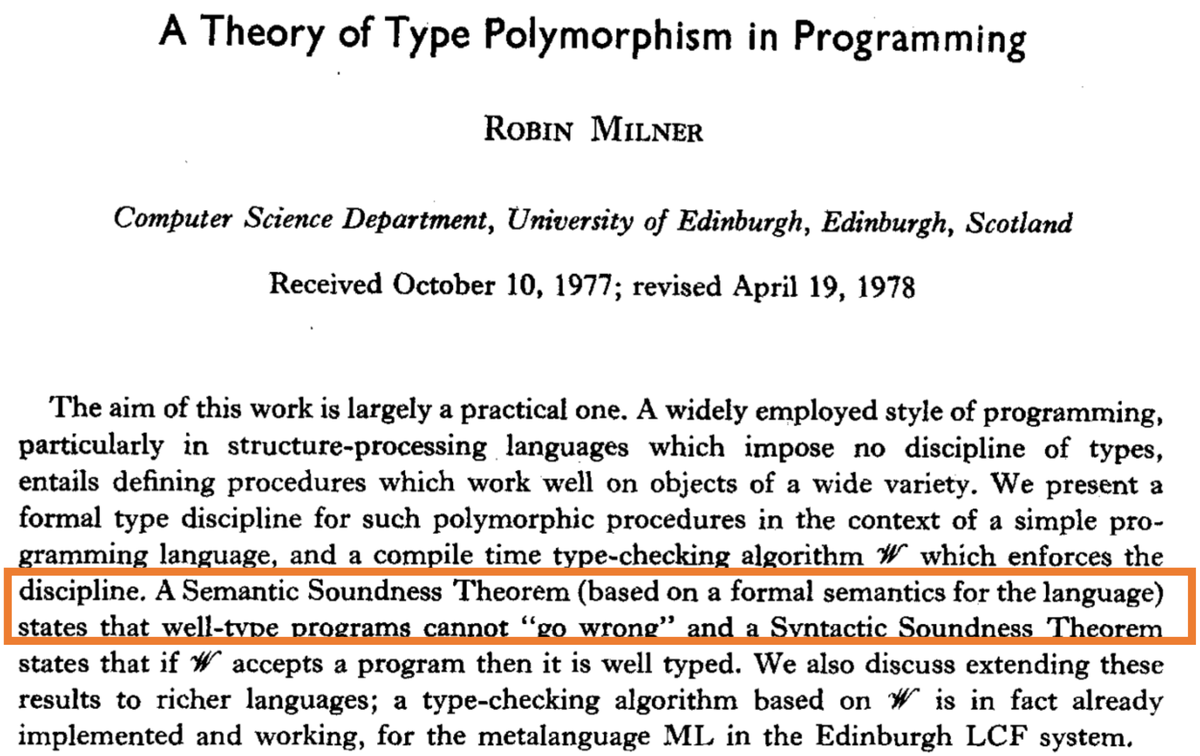
undeclared variable as l-value

```
int a;
int a;
a = 1;
```

Variable re-declaration

Analysis





A Theory of Type Polymorphism in Programming

Robin Milner

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $\mathscr{W}$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

Goals of Semantic Analysis

- Check “correct” use of programming constructs

- Ensure that the program can be compiled correctly
  - Should be able to generate code for every program that passes the semantic analysis
  - The result should be a “correct” compilation

- Runtime checks are still necessary!
  - array access, null pointer, division by zero, …
  - The semantic analysis guarantees that checks will be placed correctly by the compiler
  - (Bugs are of course still possible!)

- **But also a “contract” with the programmer**

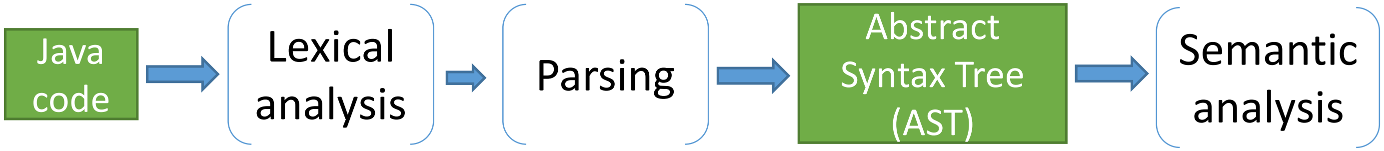
"Semantic Rules" in Java

- A variable must be declared before used
- A variable should not be declared multiple times
- A variable should be initialized before used
- Non-void method should contain return statement along all execution paths
- **`this`** keyword cannot be used in static method
- Typing and subtyping rules

Beyond Semantic Analysis

- Infer runtime properties of the program
- Whenever the execution reaches point p, the variable x cannot be used
- The value of the variable x is always positive at point p
- The pointer p cannot have a null value at point p
- Next week

Syntactic vs. Semantic Analysis


Java code

Lexical analysis

Parsing

Abstract Syntax Tree (AST)

Semantic analysis

- Construction of AST is based on context-**free** analysis
- Semantic analysis is context-**sensitive**

```
int a; ...
a = “hello”;
```

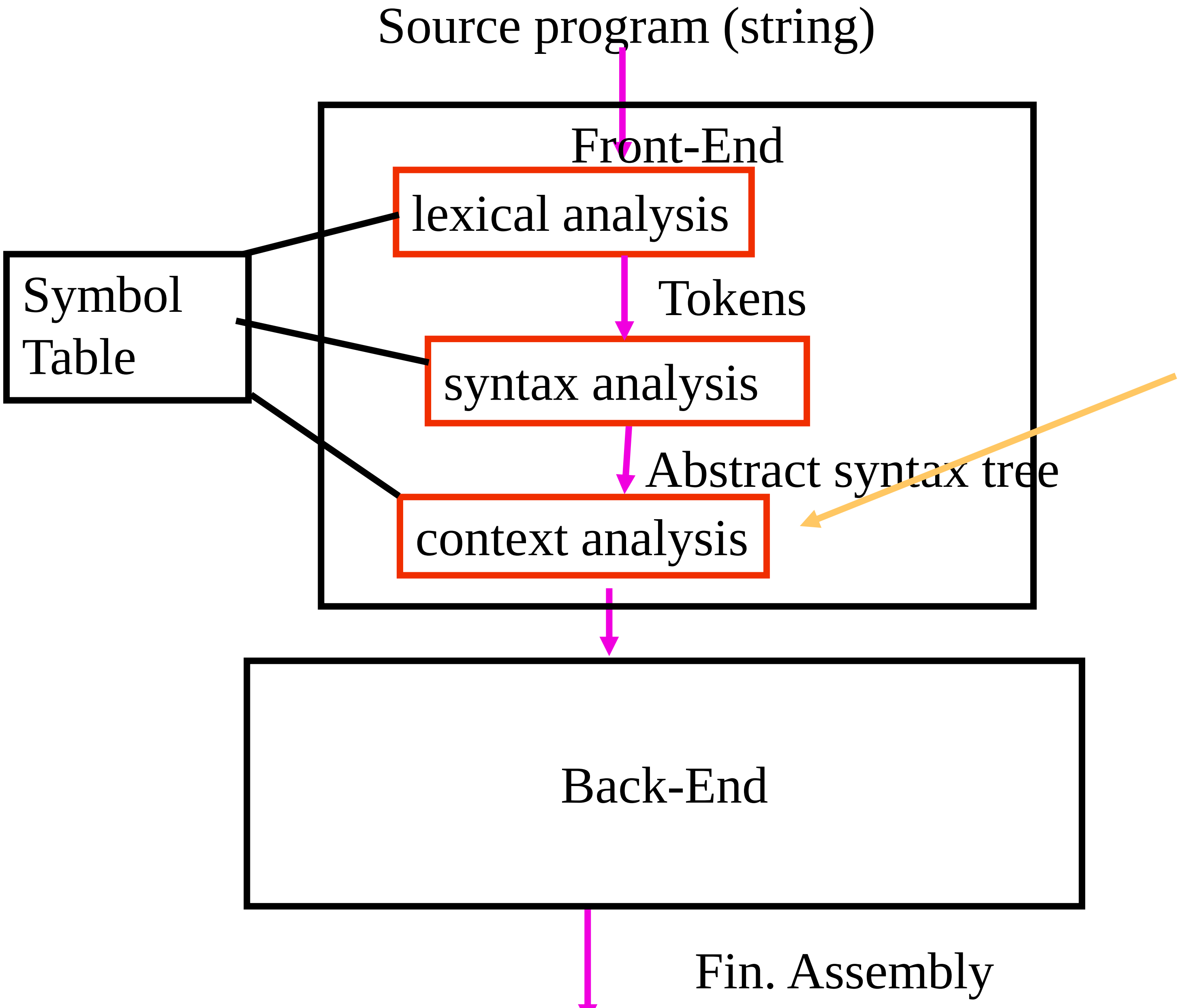
Outline

- What is Semantic (Context) Analysis
- Why is it needed?
- What is a type
- Type Checking vs. Type Inference
- A formal definition
- Scopes and type checking for imperative languages (Chapter 6)

Context Analysis

- Requirements related to the “context” in which a construct occurs
- Context sensitive requirements - cannot be specified using a context free grammar
  (Context handling)
- Requires complicated and unnatural context free grammars
- Guides subsequent phases

Basic Compiler Phases



Source program (string)

Front-End

lexical analysis

Tokens

syntax analysis

Abstract syntax tree

context analysis

Symbol Table

Back-End

Fin. Assembly

Degenerate Context Condition

- In C
  - break statements can only occur inside switch or loop statements

Partial Grammar for C

Stm → Exp;

Stm → if (**Exp**) Stm

Stm → if (**Exp**) Stm else Stm

Stm → while (**Exp**) do Stm

Stm → break;

Stm→ {StList }

StList → StList Stm

StList → ε

Refined Grammar for C

Stm→Exp;

Stm → if (**Exp**) Stm

Stm → if (**Exp**) Stm else Stm

Stm→ while (**Exp**) do LStm

Stm→ {StList }

StList → StList Stm

StList → ε

LStm → Exp;

LStm → if (**Exp**) LStm

LStm→ if (**Exp**) LStm else LStm

LStm → while (**Exp**) do LStm

LStm → {LStList }

LStm → break;

LStList → LStList LStm

LStList → ε

A Possible Abstract Syntax for C

Stmt→ Exp (Exp)
| Stmt Stmt (SeqStmt)
| Exp Stmt Stmt (IfStmt)
| Exp Stmt (WhileStmt)
| (BreakSt)

A Possible Abstract Syntax for C

```
package Absyn;
abstract public class Absyn { public int pos ;}
class Exp extends Absyn { };
class Stmt extends Absyn {} ;
class SeqStmt extends Stmt { public Stmt fstSt; public Stmt secondSt;
    SeqStmt(Stmt s1, Stmt s2) {   fstSt = s1; secondSt s2 ; }
}
class IfStmt extends Stmt { public Exp exp; public Stmt thenSt; public Stmt elseSt;
    IfStmt(Exp e, Stmt s1, Stmt s2) {   exp = e;  thenSt = s1; elseSt s2 ;  }
}
class WhileStmt extends Stmt {public Exp exp; public Stmt body;
    WhileSt(Exp e; Stmt s) { exp =e ; body = s; }
class BreakSt extends Stmt {};
```

A Context Check
(on the abstract syntax tree)

```
static void checkBreak(Stmt st)
{
     if (st instanceof SeqSt) {
         SeqSt seqst = (SeqSt) st;
         checkBreak(seqst.fstSt);  checkBreak(seqst.secondSt);
     }
     else if (st instanceof  IfSt) {
         IfSt ifst = (IfSt) st;
         checkBreak(ifst.thenSt); checkBreak(ifst elseSt);
     }
     else if (st instanceof  WhileSt) ; // skip
     else if (st instanceof  BreakeSt) {
       System.error.println(“Break must be enclosed within a loop”.
     st.pos); }
}
```

Example Context Condition: Scope Rules

- Variables must be defined within scope
- Dynamic vs. Static Scope rules
- Cannot be coded using a context free grammar

Dynamic vs. Static Scope Rules

```
 procedure  p;
              var x: integer
              procedure  q ;
                             begin { q }
                            …
                            x
                            …
                            end { q };
              procedure   r ;
              var x: integer
              begin { r }
              q ;
              end;  { r  }
begin { p }
              q ;
              r ;
end { p }
```

Summary Dynamic Rules

- Most languages enforce static rules
  - C, Java, C++, Haskel, ML, Javascript, …
- Exceptions
  - lisp, emacs
- Dynamic rules lead to ineffective compilation
- Hard to understand
  - Hinders modularity

Example Context Condition

- Types in assignment must be “compatible”'

Partial Grammar for Assignment

Stm→ id Assign Exp

Exp → IntConst

Exp → RealConst

Exp→ Exp + Exp

Exp→ Exp -Exp

Exp→ (Exp)

arg1	arg2	op	res
int	int	+, -	int
int	real	+, -	real
real	int	+, -	real
real	real	+, -	real

lhs	rhs
int	int
real	real
real	int

Refined Grammar for Assignments

Stm→ RealId Assign RealExp

Stm→IntExpAssign IntExp

Stm→RealId Assign IntExp

RealExp → RealConst

RealIntExp → RealId

RealExp→ RealExp + RealExp

RealExp→ RealExp + IntExp

RealExp→ IntExp + RealExp

RealExp→ RealExp -RealExp

RealExp→ RealExp -RealExp

RealExp→ RealExp -IntExp

RealExp→ IntExp -RealExp

RealExp→ (RealExp)

IntExp → IntConst

IntExp → IntId

IntExp→ IntExp + IntExp

IntExp→ IntExp -IntExp

IntExp→ (IntExp)

Corner Cases

- What about power operator

What is a type?

- A type is a collection of computable values that share some structural property.

Examples

```
int
string
int → bool
int × bool
```

Non-examples

```
Even integers
Positive integers
{f:int → int | x>3 =>
         f(x) > x *(x+1)}
```

Distinction between sets of values that are types and sets that are not types is *language dependent*

Advantages of Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Document intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + true – “Bill”
- Support optimization
  - Example: short integers require fewer bits
  - Access components of structures by known offset

What is a type error?

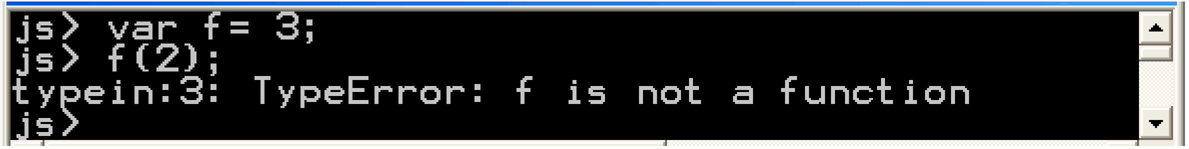
- Whatever the compiler/interpreter says it is?
- Something to do with bad bit sequences?
  - Floating point representation has specific form
  - An integer may not be a valid float
- Something about programmer intent and use?
  - A type error occurs when a value is used in a way that is inconsistent with its definition
    - Example: declare as character, use as integer

Type errors are language dependent

- Array out of bounds access
  - C/C++: run-time errors with undefined semantics
  - Java: dynamic type errors (exceptions)
- Null pointer dereference
  - C/C++: run-time errors with undefined semantics
  - Java: dynamic type errors (exceptions)
  - Rust: Compiler guarantees correctness

Compile-time vs Run-time Checking

- JavaScript and Lisp use run-time type checking
  - f(x) Make sure f is a function before calling f



```
js> var f= 3;
js> f(2);
typein:3: TypeError: f is not a function
js>
```

- Java uses compile-time type checking
  - f(x) Must have f: A → B and x : A
- Basic tradeoff
  - Both kinds of checking prevent type errors
  - Run-time checking slows down execution
  - Compile-time checking restricts program flexibility
    - JavaScript array: elements can have different types
  - Which gives better programmer diagnostics?

Expressiveness

- In JavaScript, we can write a function like

```
function f(x) { return x < 10 ? x : x(); }
```

Some uses will produce type error, some will not

- Static typing always conservative

```
   if   (complicated-boolean-expression)
   then   f(5);
    else   f(15);
```

Type Safety

- Type safe programming languages protect its own abstractions
- Type safe programs cannot go wrong
- No run-time errors
- But exceptions are fine
- The semantics of the program cannot get stuck
- Type safety is proven at language design time

Relative Type-Safety of Languages

- Not safe: Assembly, C and C++
  - Casts, unions, pointer arithmetic, …
- Almost safe: Algol family, Pascal, Ada
  - Dangling pointers
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p
    - Hard to make languages with explicit deallocation of memory fully type-safe
- Safe: Lisp, Smalltalk, ML, Haskell, Java, JavaScript
  - Dynamically typed: Lisp, Smalltalk, JavaScript
  - Statically typed: OCaml, Haskell, Java, Rust

If code accesses data, it is handled with the type associated with the creation and previous manipulation of that data

Unsafe Features of C

- Pointer arithmetic
- Casts
- Unions
- Dangling references

Pointer Arithmetic

```
int foo(){
    int a, b;
    int *p = &a;
    scanf("%d", &b);
    *(p+b) = 5;
}
```

Unions

```
int foo() {
 union {
        int i;
        int* p;
       } u;
u.i = 8;
printf("%d", *(u.p));
return 0;
}
```

Dangling References

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

Type Checking vs. Type Inference

- Standard type checking:

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Ex
  - Use declared types to check agreement
- Type inference:
  - E
  - I

~~int~~ f(~~int~~ x) { return x+1; };
~~int~~ g(~~int~~ y) { return f(y+1)*2; };

ML and Scala are *designed* to make type inference feasible

The Type Inference Problem

- Input: A program without types
- Output: A program with type for every expression
  - Every expression is annotated with its most general type

Type Checking (Imperative languages)

- Identify the type of every expression
- Usually one or two passes over the syntax tree
- Handle scope rules

Types

- What is a type
  - Varies from language to language
- Consensus
  - A set of values
  - A set of operations
- Classes
  - One instantiation of the modern notion of types

Why do we need type systems?

- Consider assembly code
  - add $r1, $r2, $r3
- What are the types of $r1, $r2, $r3?

Types and Operations

- Certain operations are legal for values of each type
  - It does not make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values because nothing else will!
- The goal of type inference is to infer a unique type for every "valid expression"

Type Checking Overview

- Three kinds of languages
  - Statically typed: (Almost) all checking of types is done as part of compilation
    - Context Analysis
    - C, Java, ML
  - Dynamically typed: Almost all checking of types is done as part of program execution
    - Code generation
    - Scheme, Python
  - Untyped
    - No type checking (Machine Code)

Type Wars

- Competing views on static vs. dynamic typing
- Static typing proponents say:
  - Static checking catches many programming errors
  - Prove properties of your code
  - Avoids the overhead of runtime type checks
- Dynamic typing proponents say
  - Static type systems are restrictive
  - Rapid prototyping difficult with type systems
  - Complicates the programming language and the compiler
  - Compiler optimizations can hide costs

Type Wars (cont.)

- In practice, most code is written in statically typed languages with escape mechanisms
  - Unsafe casts in C Java
  - union in C
  - Unsafe libraries in Rust
- It is debatable whether this compromise represents the best or worst of both worlds

Soundness of type systems

- For every expression e,
  - for every value v of e at runtime
    - $v \in val(type(e))$
- The type may actually describe more values
- The rules can reject logically correct programs
- Becomes more complicated with subtyping (inheritance)

A formal definition of type systems

Types and Programming Languages

Benjamin C. Pierce

Type judgments

- e : T
  - e is a well-typed expression of type T

- Examples
  - 2 : int
  - 2 * (3 + 4) : int
  - true : bool
  - “Hello” : string

Type judgments

- $E \vdash e : T$
  - In the context E, e is a well-typed expression of T

- Examples:
  - $b{:}\text{bool}, x{:}\text{int} \vdash b{:}\text{bool}$
  - $x{:}\text{int} \vdash 1 + x < 4{:}\text{bool}$
  - $\text{foo}{:}\text{int->string}, x{:}\text{int} \vdash \text{foo}(x) : \text{string}$

Typing rules

$$\frac{\text{Premise}}{\text{Conclusion}}\ \textbf{[Name]}$$

$$\frac{}{\text{Conclusion}}\ \textbf{[Name]}$$

Typing rules for expressions

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}} \; \mathbf{[+]}$$

Expression rules

$$\frac{\text{v: bool} \in \text{E}}{\text{E} \vdash \text{v : bool}} \qquad \frac{\text{v: int} \in \text{E}}{\text{E} \vdash \text{v : int}}$$

$$\frac{}{\text{E} \vdash \text{true : bool}} \qquad \frac{}{\text{E} \vdash \text{false : bool}}$$

$$\frac{}{\text{E} \vdash \textit{int-literal} \text{ : int}} \qquad \frac{}{\text{E} \vdash \textit{string-literal} \text{ : string}}$$

$$\frac{\text{E} \vdash \text{e1 : int} \qquad \text{E} \vdash \text{e2 : int}}{\text{E} \vdash \text{e1 } \textit{op} \text{ e2 : int}} \qquad \textit{op} \in \{ +, -, /, *, \% \}$$

$$\frac{\text{E} \vdash \text{e1 : int} \qquad \text{E} \vdash \text{e2 : int}}{\text{E} \vdash \text{e1 } \textit{rop} \text{ e2 : bool}} \qquad \textit{rop} \in \{ <=, <, >, >= \}$$

More expression rules

$$\frac{E \vdash e1 : \text{bool} \qquad E \vdash e2 : \text{bool}}{E \vdash e1\ \mathit{lop}\ e2 : \text{bool}} \qquad \mathit{lop} \in \{\ \texttt{\textbf{\&\&}},\texttt{\textbf{||}}\ \}$$

$$\frac{E \vdash e1 : \text{int}}{E \vdash \texttt{\textbf{-}}\ e1 : \text{int}} \qquad \frac{E \vdash e1 : \text{bool}}{E \vdash \texttt{\textbf{!}}\ e1 : \text{bool}}$$

$$\frac{E \vdash e1 : T[]}{E \vdash e1.\texttt{\textbf{length}} : \text{int}} \qquad \frac{E \vdash e1 : T[] \qquad E \vdash e2 : \text{int}}{E \vdash e1[e2] : T} \qquad \frac{E \vdash e1 : \text{int}}{E \vdash \texttt{\textbf{new}}\ T[e1] : T[]}$$

$$\frac{}{E \vdash \texttt{\textbf{new}}\ T() : T}$$

Subtyping

- Inheritance induces subtyping relation ≤
  - S ≤ T $\Rightarrow$ values(S) $\subseteq$ values(T)
  - “A value of type S may be used wherever a value of type T is expected”

Subtyping

- For all types:

$$\frac{}{A \leq A}$$

- For reference types:

$$\frac{A\ \mathbf{\texttt{extends}}\ B\ \{\ldots\}}{A \leq B} \qquad \frac{A \leq B \qquad B \leq C}{A \leq C} \qquad \frac{}{\mathbf{\texttt{null}} \leq A}$$

Examples

1. *int ≤ int ?*
2. *null ≤ A ?*
3. *null ≤ string ?*
4. *string ≤ null ?*
5. *null ≤ boolean ?*
6. *null ≤ boolean[] ?*
7. *A[] ≤ B[] ?*

Expression rules with subtyping

$$\frac{E \vdash e1 : T1 \quad E \vdash e2 : T2 \qquad T1 \leq T2 \text{ or } T2 \leq T1 \qquad op \in \{==, !=\}}{E \vdash e1 \ op \ e2 : bool}$$

Rules for method invocations

$$\frac{E \vdash e_0 : T_1 \times \ldots \times T_n \to T_r \qquad E \vdash e_i : T'_i \quad T'_i \leq T_i \text{ for all } i=1..n}{E \vdash e_0(e_1, \ldots, e_n) : T_r}$$

$$\frac{(m : \text{static } T_1 \times \ldots \times T_n \to T_r) \in C \qquad E \vdash e_i : T'_i \quad T'_i \leq T_i \text{ for all } i=1..n}{E \vdash c.m(e_1, \ldots, e_n) : T_r}$$

Statement rules

- Statements have type **`void`**
- Judgments of the form

$$E \vdash S$$

  - In environment E, S is well-typed

$$\frac{E \vdash e:\text{bool} \qquad E \vdash S}{E \vdash \textbf{\texttt{while}}\ (e)\ S} \qquad \frac{E \vdash e:\text{bool} \qquad E \vdash S}{E \vdash \textbf{\texttt{if}}\ (e)\ S} \qquad \frac{E \vdash e:\text{bool} \qquad E \vdash S_1 \quad E \vdash S_2}{E \vdash \textbf{\texttt{if}}\ (e)\ S_1\ \text{else}\ S_2}$$

$$\frac{}{E \vdash \textbf{\texttt{break}}} \qquad \frac{}{E \vdash \textbf{\texttt{continue}}}$$

Return statements

- $\texttt{\textbf{ret}}:T_r$ represents return type of current method

$$\frac{E \vdash e:T \quad \texttt{\textbf{ret}}:T' \in E \quad T \leq T'}{E \vdash \texttt{\textbf{return e;}}}$$

$$\frac{\texttt{\textbf{ret}}:\text{void} \in E}{E \vdash \texttt{\textbf{return;}}}$$

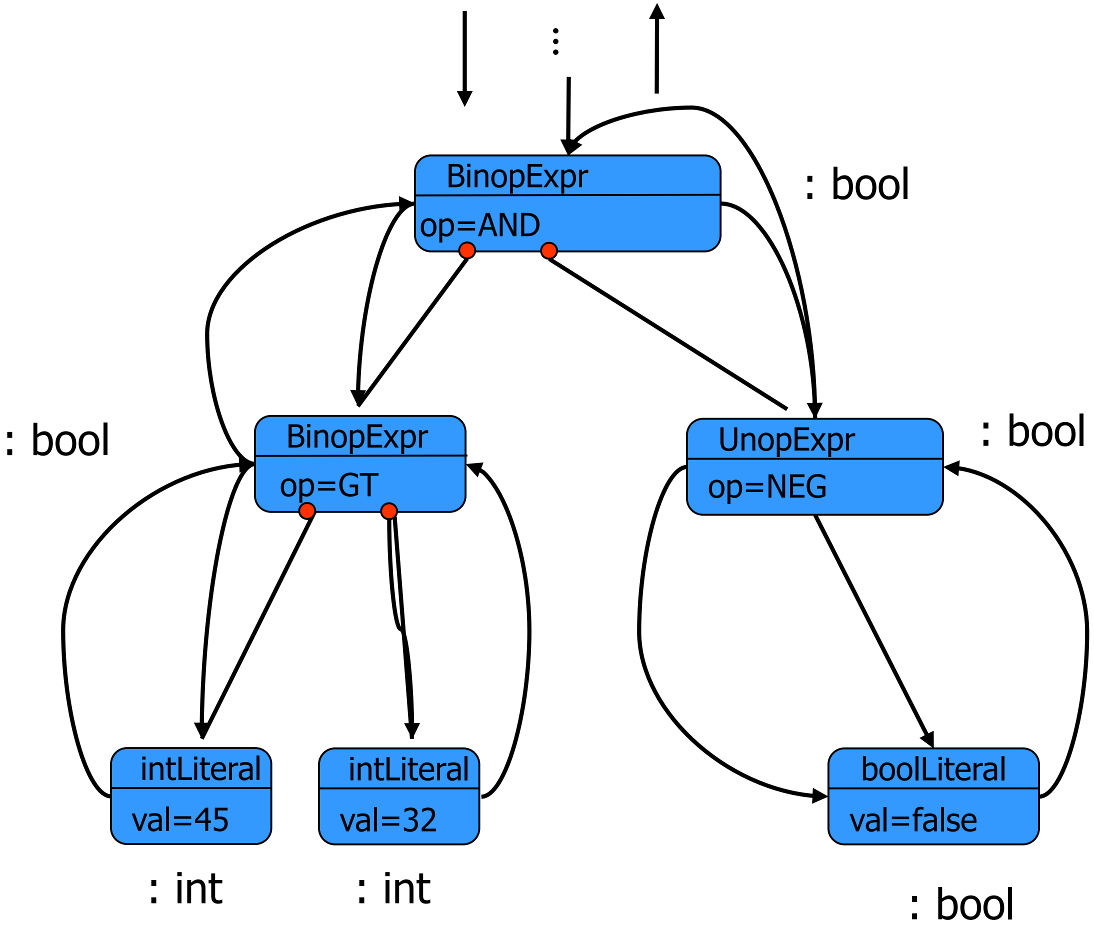
Type-checking algorithm

1. Construct types
    1. Add basic types to a “**type table”**
    2. Traverse AST looking for user-defined types (classes,methods,arrays) and store in table
    3. Bind all symbols to types

Type-checking algorithm

2. Traverse AST bottom-up (using visitor)
    1. For each AST node find corresponding rule (there is only one for each kind of node)
    2. Check if rule holds
        1. **Yes:** assign type to node according to consequent
        2. **No:** report error

Algorithm example



$$\frac{E \vdash e1 : \text{bool} \qquad E \vdash e2 : \text{bool}}{E \vdash e1 \ \textit{\&\&}\ e2 : \text{bool}}$$

$$\frac{E \vdash e1 : \text{bool}}{E \vdash !e1 : \text{bool}}$$

$$\frac{E \vdash e1 : \text{int} \qquad E \vdash e2 : \text{int}}{E \vdash e1 > e2 : \text{bool}}$$

$$\frac{}{E \vdash \texttt{false} : \text{bool}}$$

$$\frac{}{E \vdash \textit{int-literal} : \text{int}}$$

`45 > 32 && !false`

Type Safety Formally

- A program is typeable if there exists a derivation of the types using the inference rules
- A programming language is type safe with respect to a type system if every typable program cannot go wrong
  - No undefined behavior
  - An interpreter will not get stuck
  - A compiler will generate code w/o undefined behavior

Eiffel, 1989

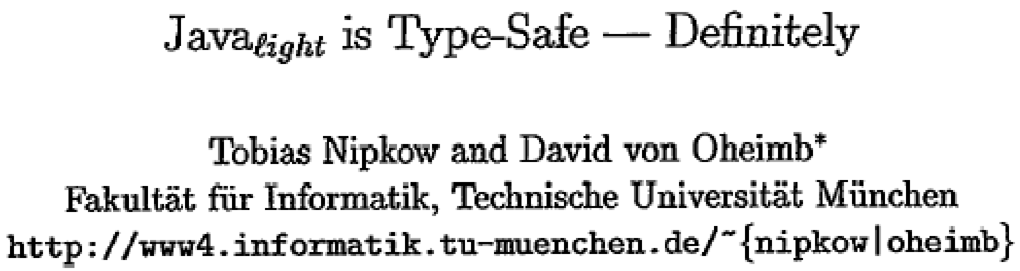
Cook, W.R. (1989) - *A Proposal for Making Eiffel Type-Safe*, in Proceedings of ECOOP'89. S. Cook (ed.), pp. 57-70. Cambridge University Press.

Betrand Meyer, on unsoundness of Eiffel:
“Eiffel users universally report that they almost never run into such problems in real software development.”

Ten years later: Java

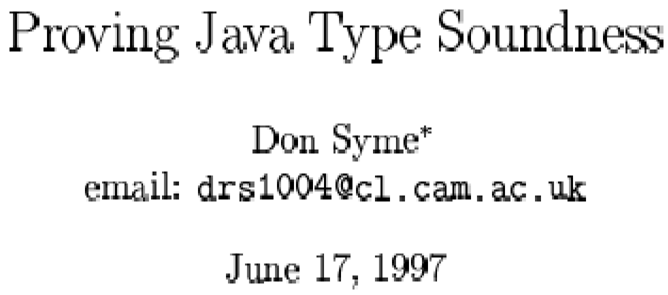
Java is not type-safe

Vijay Saraswat
AT&T Research, 180 Park Avenue, Florham Park NJ 07932



Java$_{light}$ is Type-Safe — Definitely

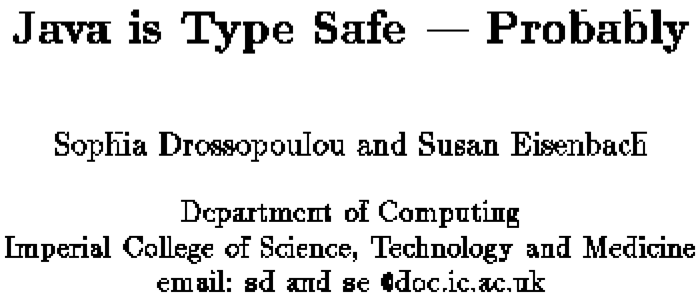
Tobias Nipkow and David von Oheimb*
Fakultät für Informatik, Technische Universität München
`http://www4.informatik.tu-muenchen.de/~{nipkow|oheimb}`



Proving Java Type Soundness

Don Syme*
email: `drs1004@cl.cam.ac.uk`

June 17, 1997



Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Twenty years later: Java + Generics

Yossi Gil, Tomer Levy:
Formal Language Recognition with the Java Type Checker.
ECOOP 2016: 10:1-10:27

Radu Grigore:
Java generics are turing complete.
POPL 2017: 73-85

Nada Amin, Ross Tate:
Java and scala's type systems are unsound: the existential crisis of null pointers.
OOPSLA 2016: 838-848

Type Checking Implementation

Type Checking Implementation

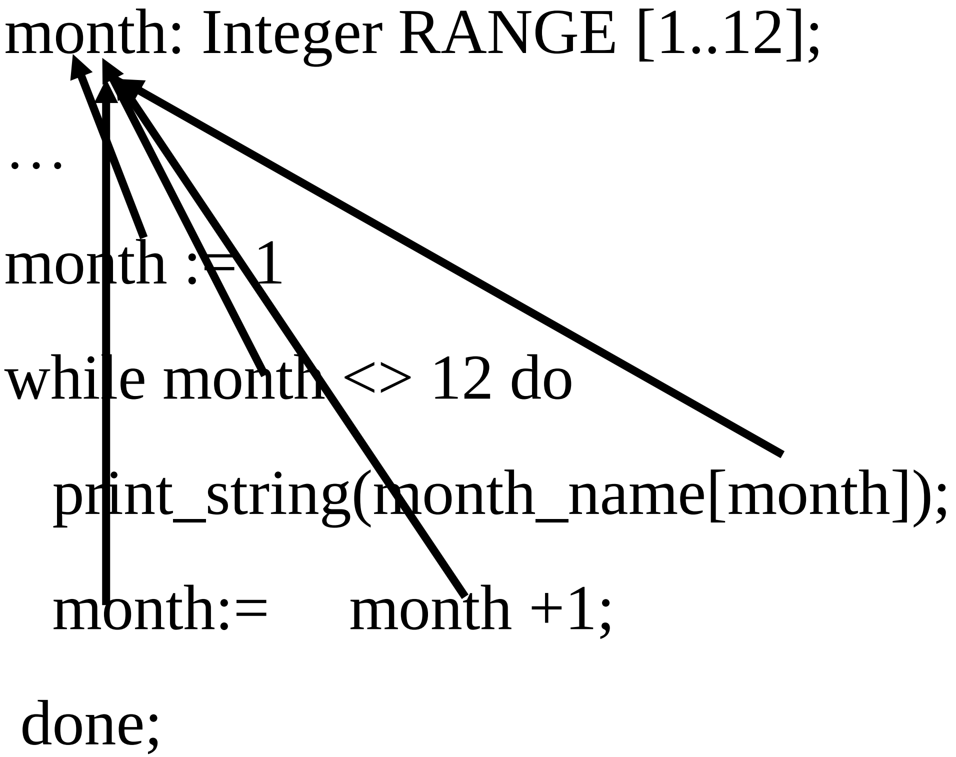
- Multiple AST traversals
- Permit use before definition
- Creates a symbol table and class table

Issues in Context Analysis Implementation

- Name Resolution
- Type Checking
  - Type Equivalence
  - Type Coercions
  - Casts
  - Polymorphism
  - Type Constructors

Name Resolution (Identification)

- Connect applied occurrences of an identifier/operator to its defining occurrence



```
month: Integer RANGE [1..12];
…
month := 1
while month <> 12 do
   print_string(month_name[month]);
   month:=     month +1;
 done;
```

Name Resolution (Identification)

- Connect applied occurrences of an identifier/operator to its defining occurrence
- Forward declarations
- Separate name spaces
- Scope rules

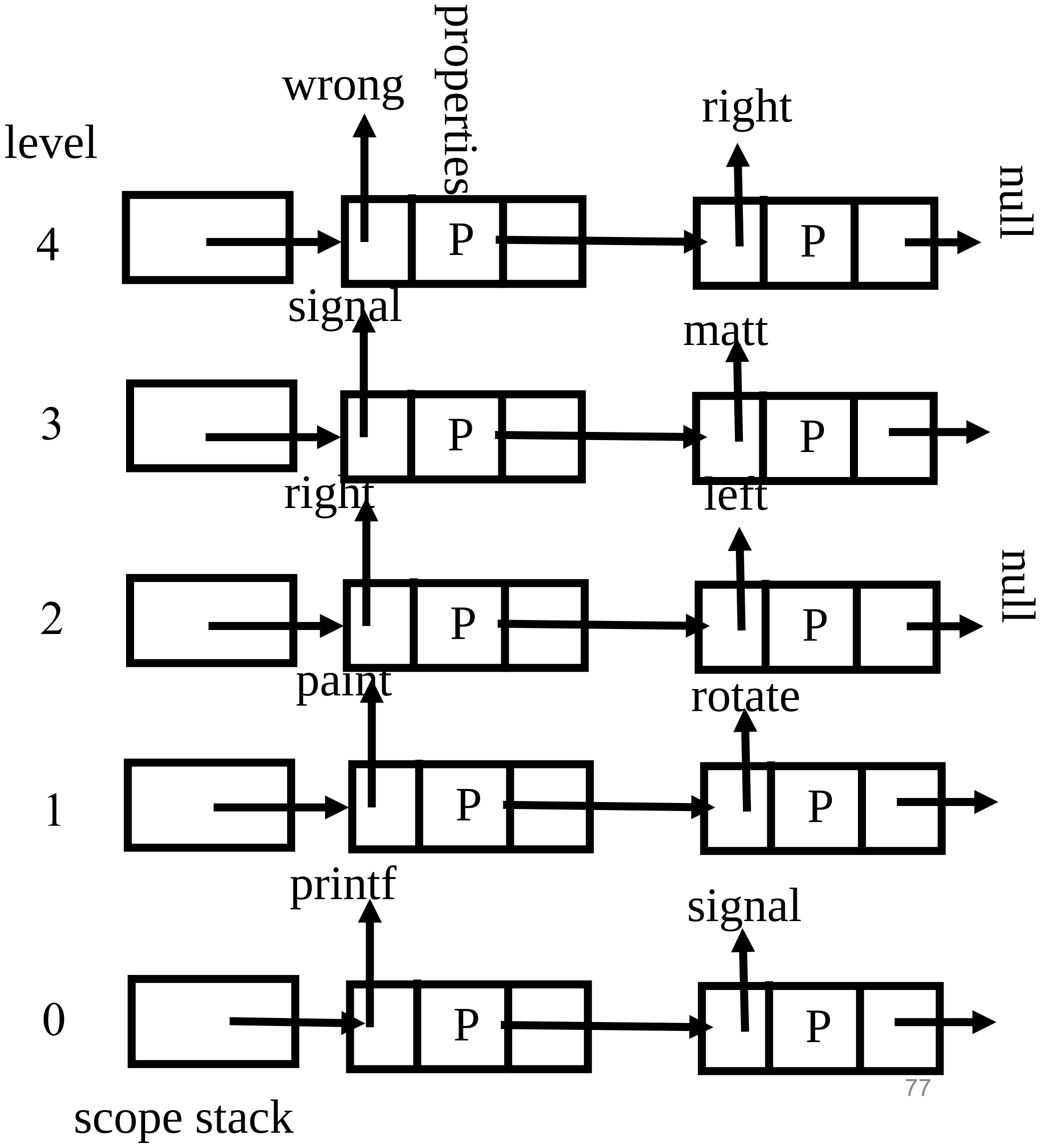
```
struct one_int {
    int i ;
} i;
 i.i = 3;
```

A Simple Implementation

- A separate table per scope/name space
- Record properties of identifiers
- Create entries for defining occurrences
- Search for entries for applied occurrences
- Create table per scope enter
- Remove table per scope enter
- Expensive search

Example

```
void roate(double angle) {
…
}
void paint(int left, int right) {
  Shade matt, signal;
…
    {
     Counter right; wrong ;
     …
    }
}
```

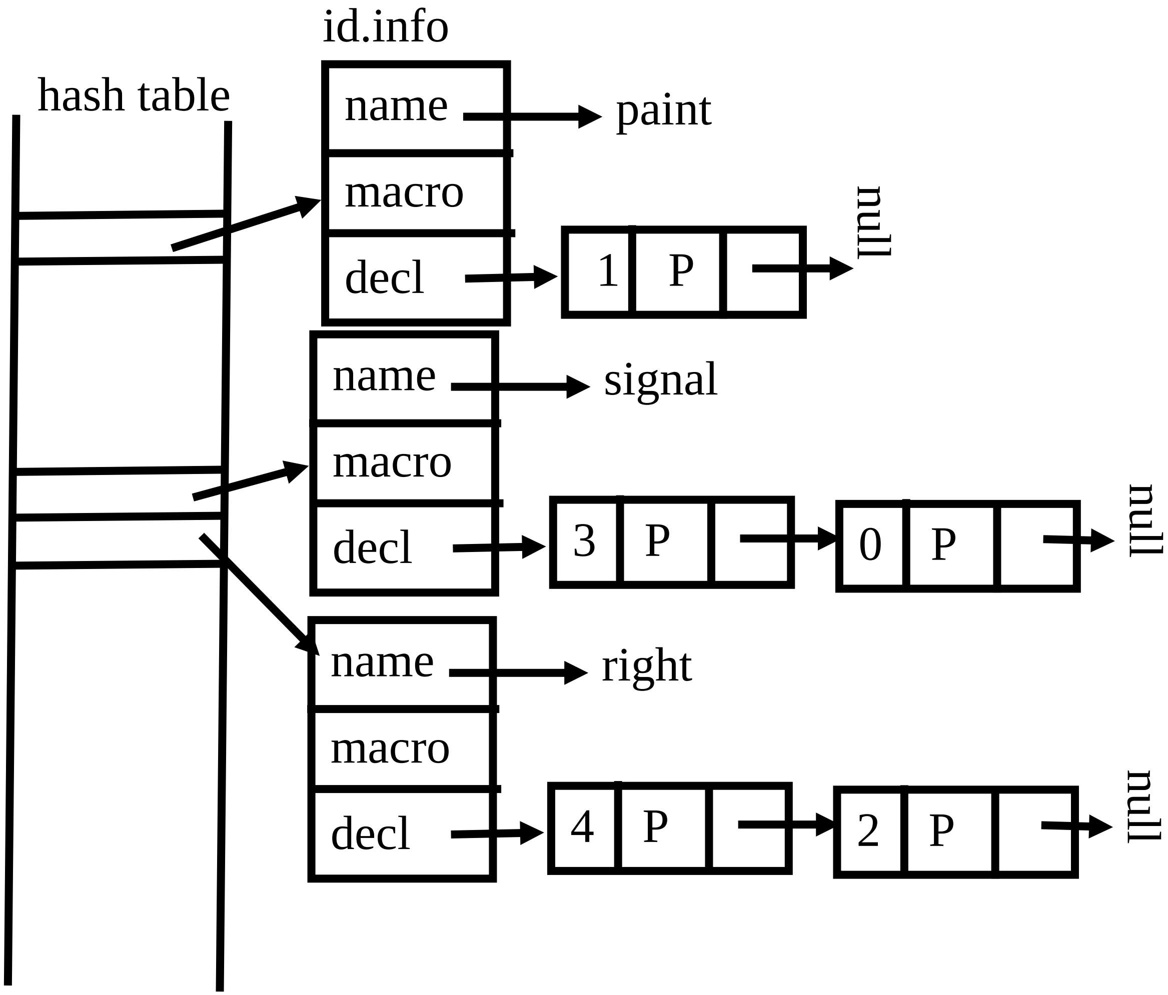


A Hash-Table Based Implementation

- A unified hashing table for all occurrences
- Separate entries for every identifier
- Ordered lists for different scopes
- Separate table maps scopes to the entries in the hash
  - Used for ending scopes

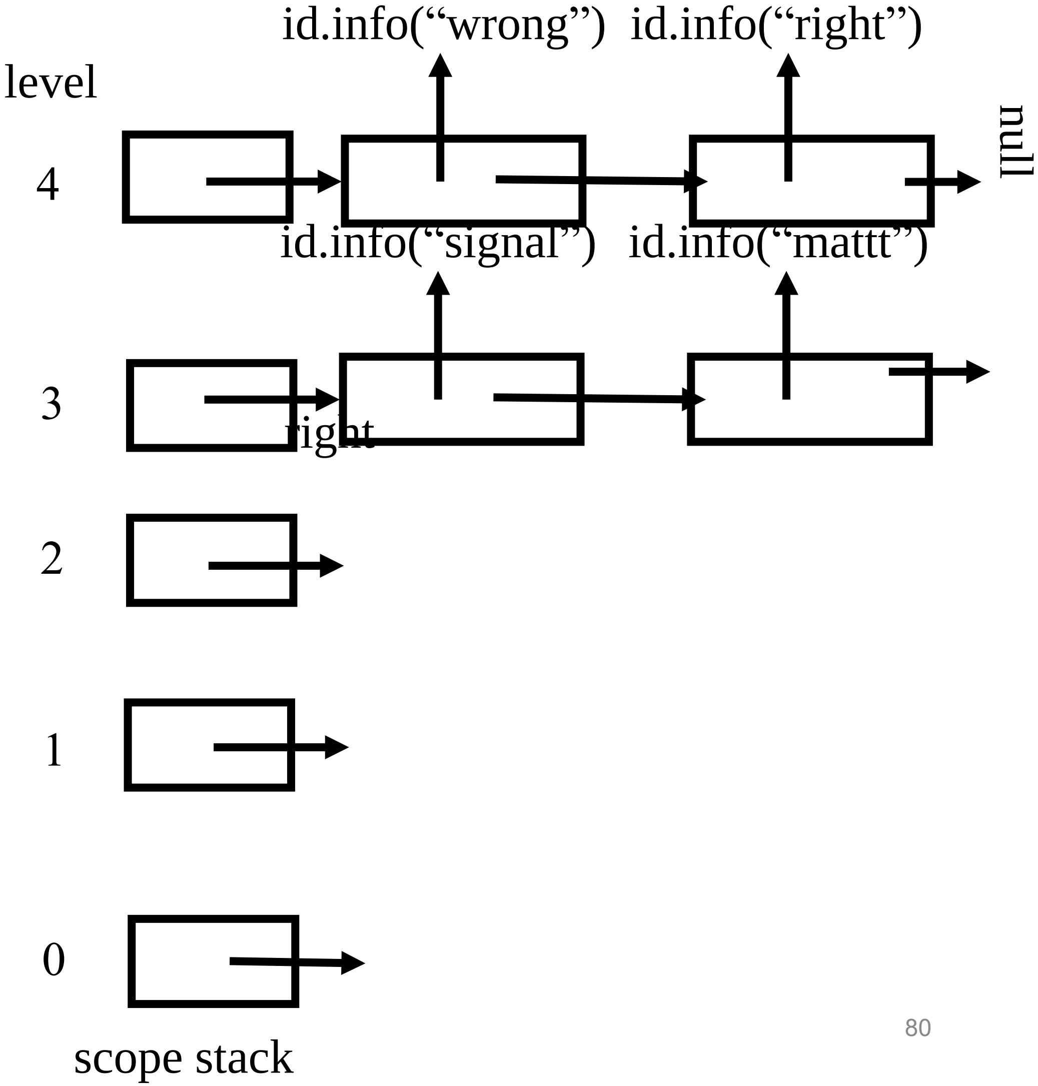
Example

```
void roate(double angle) {
…
}
void paint(int left, int right) {
  Shade matt, signal;
…
    {
     Counter right; wrong ;
     …
    }
}
```



Example(cont.)

```
void roate(double angle) {
…
}
void paint(int left, int right) {
  Shade matt, signal;
…
    {
     Counter right; wrong ;
     …
    }
}
```



Overloading

- Some programming languages allow to resolve identifiers based on the context
  - 3 + 5 is different than 3.1 + 5.1
- Overloading user defined functions
  PUT(s: STRING) PUT(i: INTEGER)
- Type checking and name resolution interact
- May need several passes

Type Equivalence

- Name equivalence
  - TYPE t1 = ARRAY[Integer] of Integer;
  - TYPE t2 = ARRAY[Integer] of Integer;
  - TYPE t3 = ARRAY[Integer] of Integer;
  - TYPE t4 = t3;
- Structural equivalence
  - TYPE t5= RECORD {c: Integer ; p: Pointer to t5;}
  - TYPE t6= RECORD {c: Integer ; p: Pointer to t6 ;}
  - TYPE t7 = RECORD {c: Integer ; p : Pointer to
    RECORD {c: Integer ; p: Pointer to t5;}}

Casts and Coercions

- The compiler may need to insert implicit conversions between types

  `float x = 5;`
- The programmer may need to insert explicit conversions between types

Kind Checking

Defined L-values in assignments

expected

found

	lvalue	rvalue
lvalue	-	deref
rvalue	error	-

Type Constructors

- Record types
- Union Types
- Arrays

Arrays and Subtyping Can break type safety

Array of strings ≤ Array of Any ?

```
 Array[String] x = new Array[String](1);
 Array[Any] y= x;
y.set(0, new FooBar());
// just stored a FooBar in a String array!
```

Routine Types

- Usually not considered as data
- The data can be a pointer to the generated code

Generics

- Enable reuse of code
- Types as “Variables”
- Generalize overloading

Generic Example

```
// generic method printArray
public static <E> void printArray( E[] inputArray ) {
   // Display array elements
   for(E element : inputArray) {
      System.out.printf("%s ", element);
   }
   System.out.println();
}
```

Generic Example (use)

```
public static void main(String args[]) {
     // Create arrays of Integer, Double and Character
     Integer[] intArray = { 1, 2, 3, 4, 5 };
     Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

     System.out.println("Array integerArray contains:");
     printArray(intArray);   // pass an Integer array

   System.out.println("\nArray characterArray contains:");
     printArray(charArray);   // pass a Character array
  }
}
```

Dynamic Checks

- Certain consistencies need to be checked at runtime in general
- But can be statically checked in many cases
- Examples
  - Overflow
  - Bad pointers
  - Array out of bounds
  - Safe downcasts

Summary

- Semantics checks ensure good properties of program
- Defined by the programming languages
- Tradeoffs
  - Security
  - Ease of use
  - Efficiency of generated code
  - Expressive power
  - Reusability
- Implemented via multiple passes on the AST