Check-In 16

Review: Type Systems

Give an example of a C program that uses type coercion

Check-In 16: Solution

Review: Type Systems

Administrivia

Housekeeping

- P3 deadline tonight
- P4 released “Monday morning”, i.e. Sunday @ 11:59 PM + 1 minute

University of Kansas| Drew Davidson

EECS 665
COMPILER CONSTRUCTION

Type Analysis

Last Lecture

Review: Type Systems

Discuss Type Systems

- What they are
- Why we use them

Type Specification (optional)

- How we communicate type systems

You Should Know

- What a type system is
- How type systems effect semantics

Semantics

Today's Outline

Type Analysis

Enforcing Type Systems

- Design points

Type Analysis

- Type checking
- Type inference / synthesis

Semantics

Enforcing Type Systems

Language property: how much enforcement / checking to do?

- Idea 1: check what you can, allow uncertainty

e.g. C

- Idea 2: check what you can, disallow uncertainty completely

e.g. Haskell

- Idea 3: check what you can, force user to dispel uncertainty

e.g. Java, Rust

Escaping the Type System

Enforcing Types

Some languages allow an explicit means to “escape” the type system

- Typecasting – allow one type to be used as another type

Casting Within Hierarchy

Enforcing Types

Cross-casting (static check in Java)

```
Apple a = new Apple();
Orange o = (Orange)a;
```

Compiler check

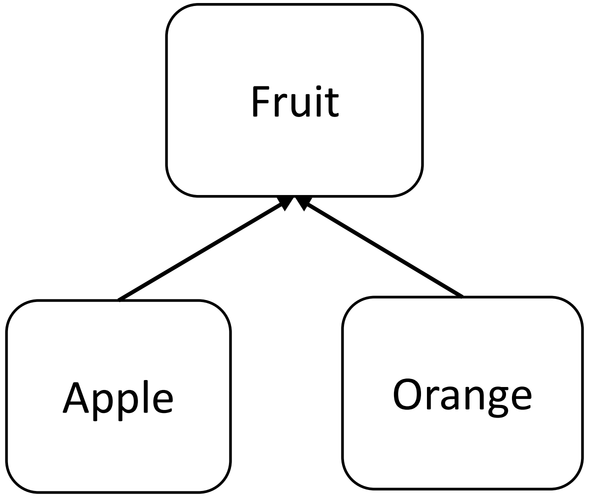
Downcasting (dynamic check in Java)

```
Fruit f = new Apple();
if ( rand() ) {
  f = new Orange();
}
Apple dApp = (Apple)f;
```

ok

Runtime check

Class Hierarchy


Fruit

Apple

Orange

Casting Within Hierarchy

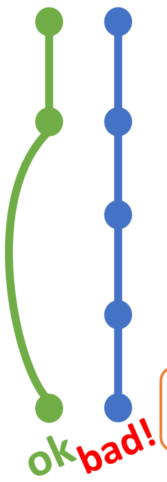
Enforcing Types

Cross-casting (static check in Java)

```
Apple a = new Apple();
Orange o = (Orange)a;
```

Compiler check

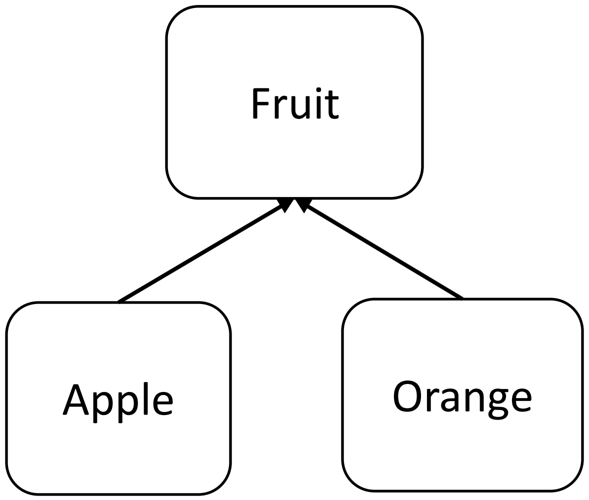
Downcasting (dynamic check in Java)



```
Fruit f = new Apple();
if ( rand() ) {
  f = new Orange();
}
Apple dApp = (Apple)f;
```

Runtime check

Class Hierarchy



Strongly-Typed vs Weakly-Typed

Enforcing Types

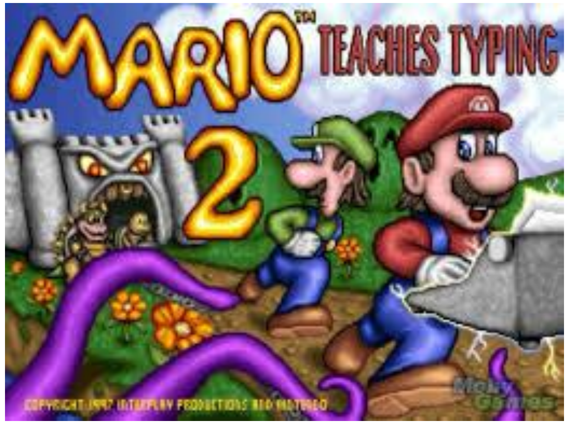
Colloquial classification of a language's type system

- Degree to which type errors are allowed to happen at runtime
- Continuum without precise definitions

Type Safety

Enforcing Types

- Has a precise definition
  - All successful operations must be allowed by the type system
- Java was explicitly designed to be type safe
  - A variable of some type can only be used as that type without causing an error
- C is very much not type safe
- C++ isn't either but it is safer



Type Safety Violations

Type Enforcement

C

Format specifier

```
printf("%s", 1);
```

Memory safety

```
struct big{
  int a[1000000];
};
struct big * b = malloc(1);
```

C++

```
Unchecked casts
class T1 { char a };
class T2 { int b };
int main {
  T1 * myT1 = new T1();
  T2 * myT2 = new T2();
  myT1 = (T1*)myT2;
}
```

Type Research

Detour: Ungraded Material

Research on Types

Type Checking

A huge topic in and of itself

- Some CS Deparments have a “PLT” focus: “Programming Languages and Types”

Liquid Types *

Patrick M. Rondon   Ming Kawaguchi   Ranjit Jhala
University of California, San Diego
{prondon,mwookawa,jhala}@cs.ucsd.edu

Abstract

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system that combines *Hindley-Milner* type inference with *Predicate Abstraction* to automatically infer dependent types precise enough to prove a variety of safety properties. Liquid types allow programmers to reap many of the benefits of dependent types, namely static verification of critical properties and the elimination of expensive run-time checks, without the heavy price of manual annotation. We have implemented liquid type inference in DSOLVE, which takes as input an OCAML program and a set of logical qualifiers and infers dependent types for the expressions in the OCAML program. To demonstrate the utility of our approach, we describe experiments using DSOLVE to statically verify the safety of array accesses on a set of OCAML benchmarks that were previously annotated with dependent types as part of the DML project. We show that when used in conjunction with a fixed set of array bounds checking qualifiers, DSOLVE reduces the amount of manual annotation required for proving safety from 31% of program text to under 1%.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Dependent Types, Hindley-Milner, Predicate Abstraction, Type Inference

1. Introduction

Modern functional programming languages, like ML and Haskell, have many features that dramatically improve programmer productivity and software reliability. Two of the most significant are strong static typing, which detects a host of errors at compile-time, and type inference, which (almost) eliminates the burden of annotating the program with type information, thus delivering the benefits of strong static typing for free.

The utility of these type systems stems from their ability to predict, at compile-time, invariants about the run-time values computed by the program. Unfortunately, classical type systems only capture relatively coarse invariants. For example, the system can express the fact that a variable `i` is of the type `int`, meaning that it is always an integer, but not that it is always an integer within a certain range, say between 1 and 99. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by `i`, or the accessing of an array `a` of size 100 at an index `i`. Instead, the language can only provide a weaker dynamic safety guarantee at the additional cost of high performance overhead.

In an exciting development, several authors have proposed the use of *dependent types* [20] as a mechanism for enhancing the expressivity of type systems [14, 27, 2, 22, 10]. Such a system can express the fact

$$\mathtt{i} :: \{\nu{:}\mathtt{int} \mid 1 \leq \nu \wedge \nu \leq 99\}$$

which is the usual type `int` together with a *refinement* stating that the run-time value of `i` is an always an integer between 1 and 99. Pfenning and Xi devised DML, a practical way to integrate such types into ML, and demonstrated that they could be used to recover static guarantees about the safety of array accesses, while simultaneously making the program significantly faster by eliminating run-time checking overhead [27]. However, these benefits came at the price of automatic inference. In the DML benchmarks, about 31% of the code (or 17% by number of lines) is manual annotations that the typechecker needs to prove safety. We believe that this non-trivial annotation burden has hampered the adoption of dependent types despite their safety and performance benefits.

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system for automatically inferring dependent types precise enough to prove a variety of safety properties, thereby allowing programmers to reap many of the benefits of dependent types without paying the heavy price of manual annotation. The heart of our inference algorithm is a technique for blending Hindley-Milner type inference with *predicate abstraction*, a technique for synthesizing loop invariants for imperative programs that forms the algorithmic core of several software model checkers [3, 16, 4, 29, 17]. Our system takes as input a program and a set of *logical qualifiers* which are simple boolean predicates over the program variables, a special *value variable* ν, and a special placeholder variable $\star$ that can be instantiated with program variables. The system then infers *liquid types*, which are dependent types where the refinement predicates are *conjunctions* of the logical qualifiers.

In our system, type checking and inference are decidable for three reasons (Section 3). First, we use a conservative but decidable notion of subtyping, where we reduce the subtyping of arbitrary dependent types to a set of implication checks over base types, each of which is deemed to hold if and only if an *embedding* of the implication into a decidable logic yields a valid formula in the logic. Second, an expression has a valid liquid type derivation only if it has a valid ML type derivation, and the dependent type

---

* This work was supported by NSF CAREER grant CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603, and a gift from Microsoft Research.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

Refinement Types

Type Checking

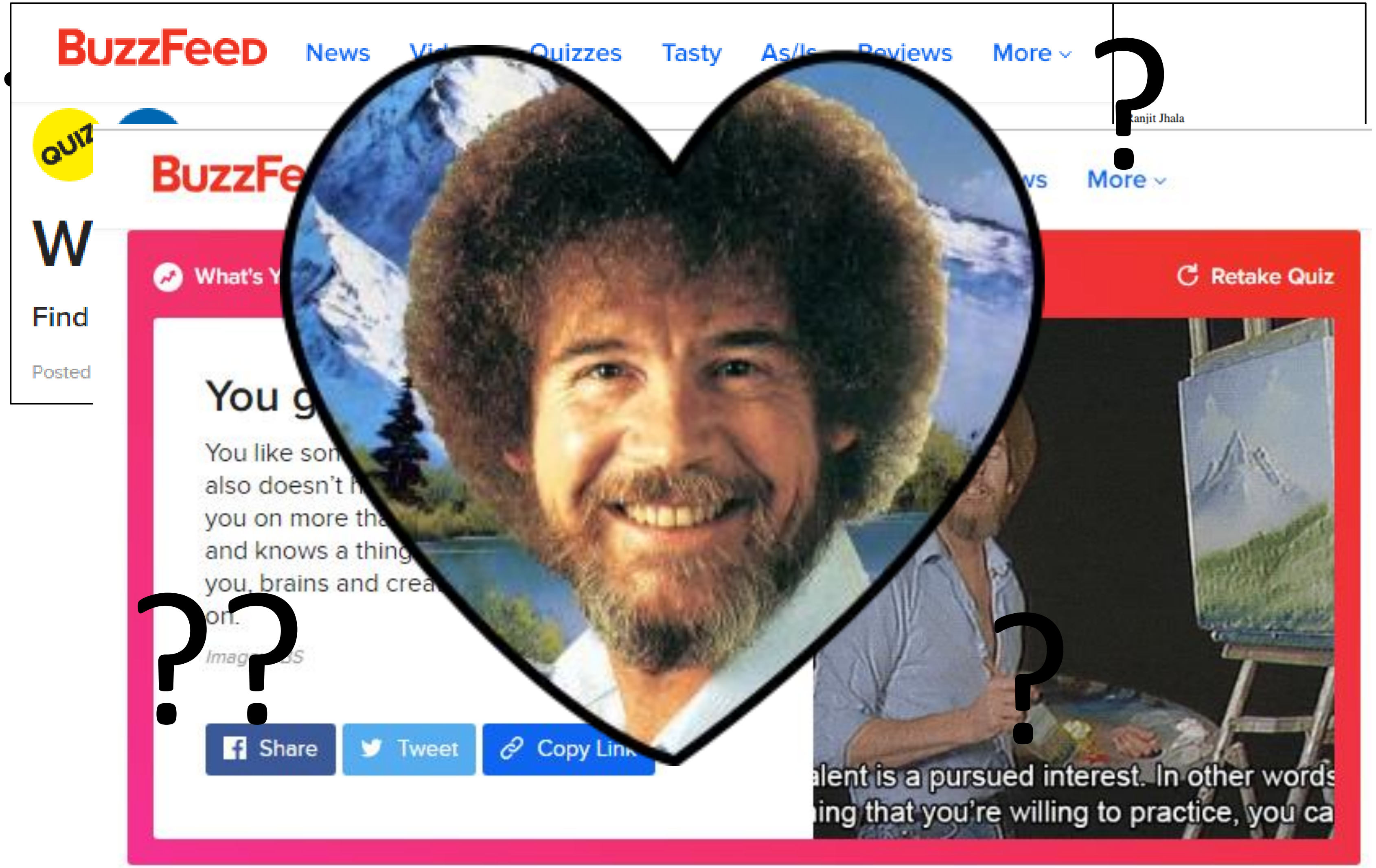
- A type enhanced with a predicate which must hold for any element of that type

$$f : \mathbb{N} \rightarrow \{ n : \mathbb{N} \mid n\%2 = 0 \}$$

- Could imagine enhancing a type system with annotations for all kinds of properties
  - Single-use variable
  - High security/low security (non-interference)

More Research on Types

Type Checking



You g

You like so
also doesn't
you on more tha
and knows a thing
you, brains and crea
on.

Imag S

alent is a pursued interest. In other words
ing that you're willing to practice, you ca

Piggybacking on Type Checking

- Type checking is a good place to get extra programmer hints:
  - Programmers are already familiar with typing logic
  - The analysis is already well-formulated

Formal Type Systems

End Detour: Done with Ungraded Material

Reasons for Typing

Type Checking

Generate appropriate code for operations

A + B

- String concatenation? Integer addition? Floating-point addition

Catch runtime errors / security

- Make sure operations are sensible
- Augment type system with addition checks

Types In Action

Type Checking

Type Analysis

- Assigning types to expressions
- Flavors:
    - Type synthesis – get type of an AST node from it's children
    - Type inference – get type of an AST node from it's use context

Type Checking

- Ensure that type of a construct is allowed by the type system

Implementing Our Type Checker

Type Checking

Implementing Typing

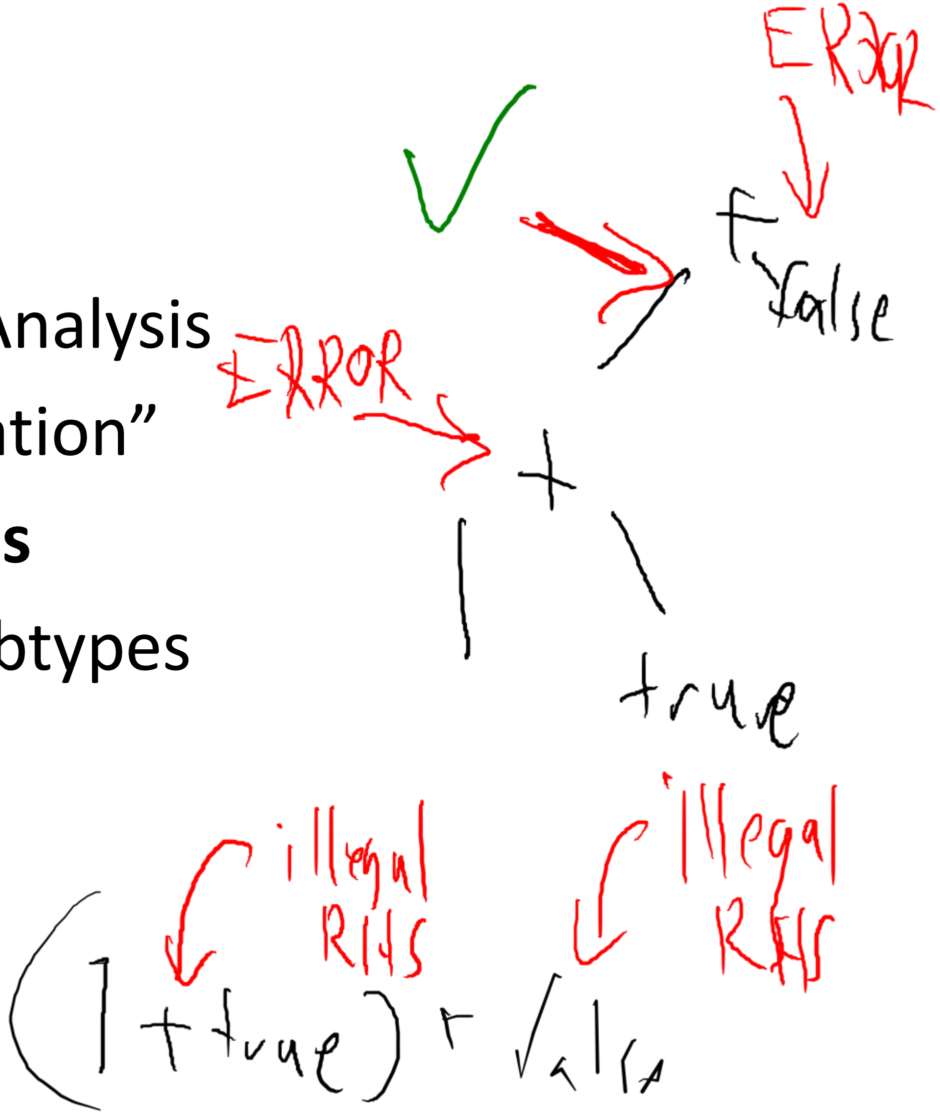
Our Type System

Structurally similar to nameAnalysis

- Historically, intermingled with nameAnalysis
- Done as part of AST attribute "decoration"

Add a typeCheck method to AST nodes

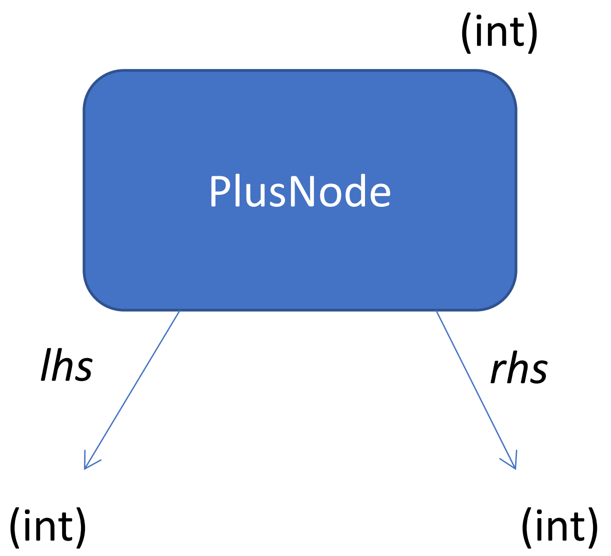
- Recursively walk the AST checking subtypes
  - "Inside out" analysis
  - Attach types to nodes
  - Propagate an error symbol



Binary Operators

Implementing Static Typing

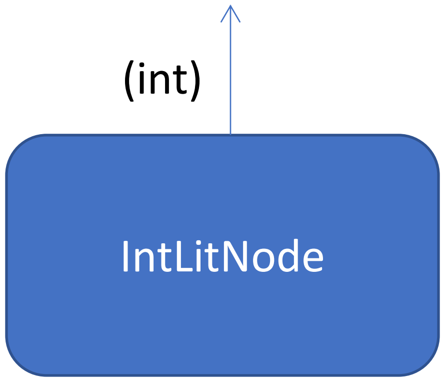
- Get the type of the LHS
- Get the type of the RHS
- Check that the types are compatible for the operator
- Set the *kind* of the node be a value
- Set the *type* of the node to be the type of the operation's result



Literals

Implementing Static Typing

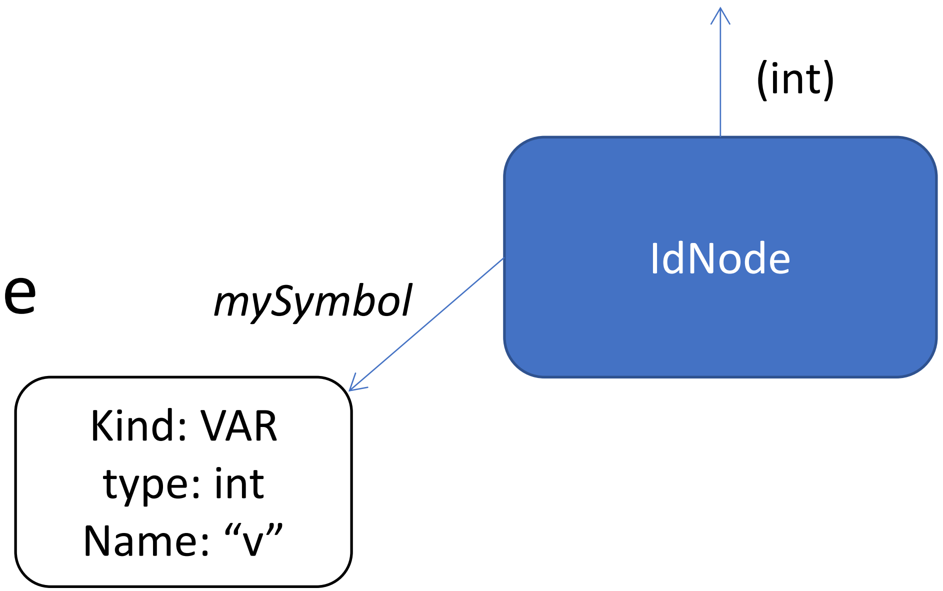
- Cannot be wrong
  - Just pass the type of the literal up the tree



Variables

Implementing Static Typing

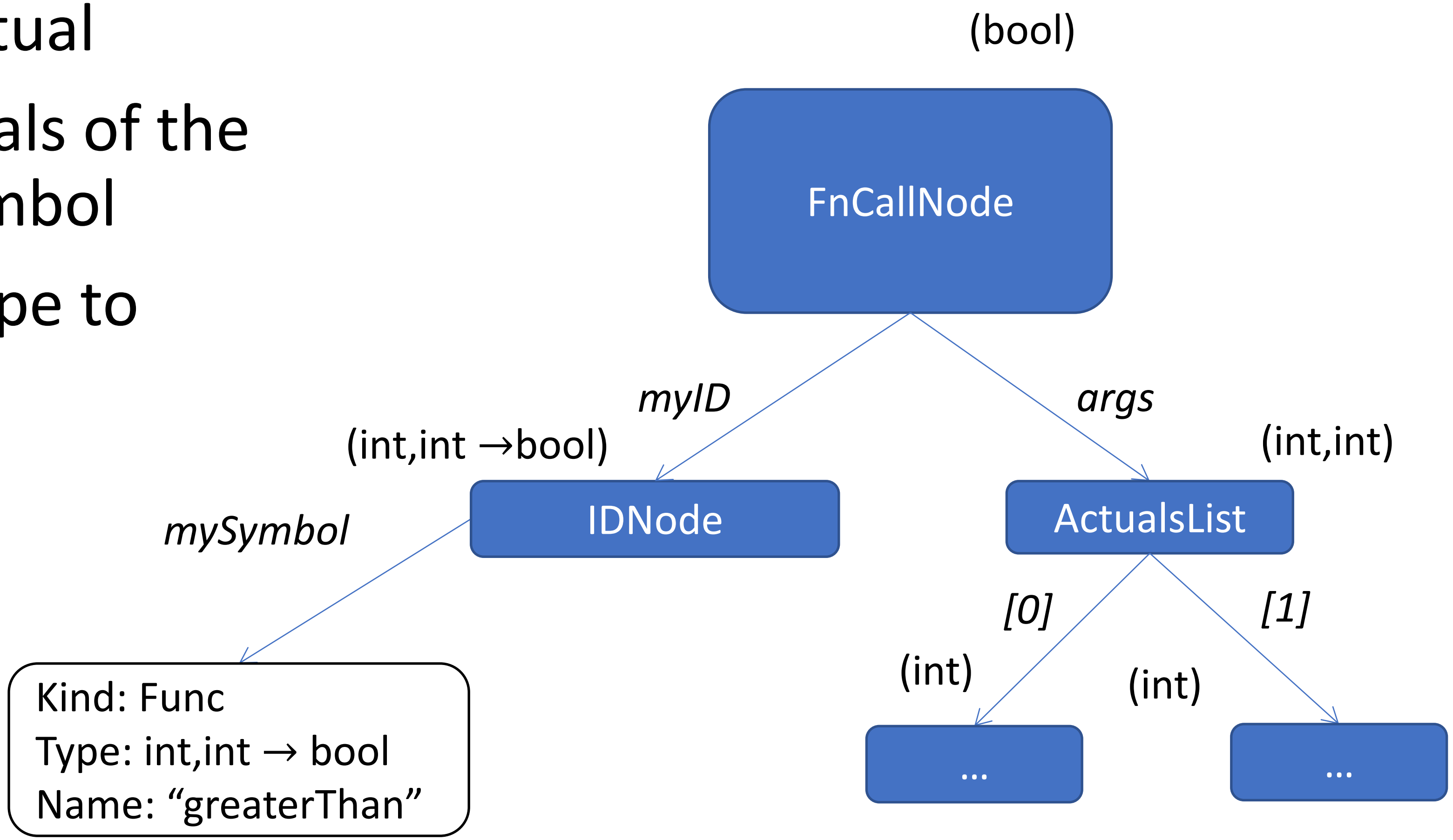
- Look up the type of the declaration
  - There should be a symbol "linked" to the node
- Pass symbol type up the tree



Function Calls

Implementing Type Checking

- Get type of each actual
- Match against formals of the called function's symbol
- Propagate return type to parent node



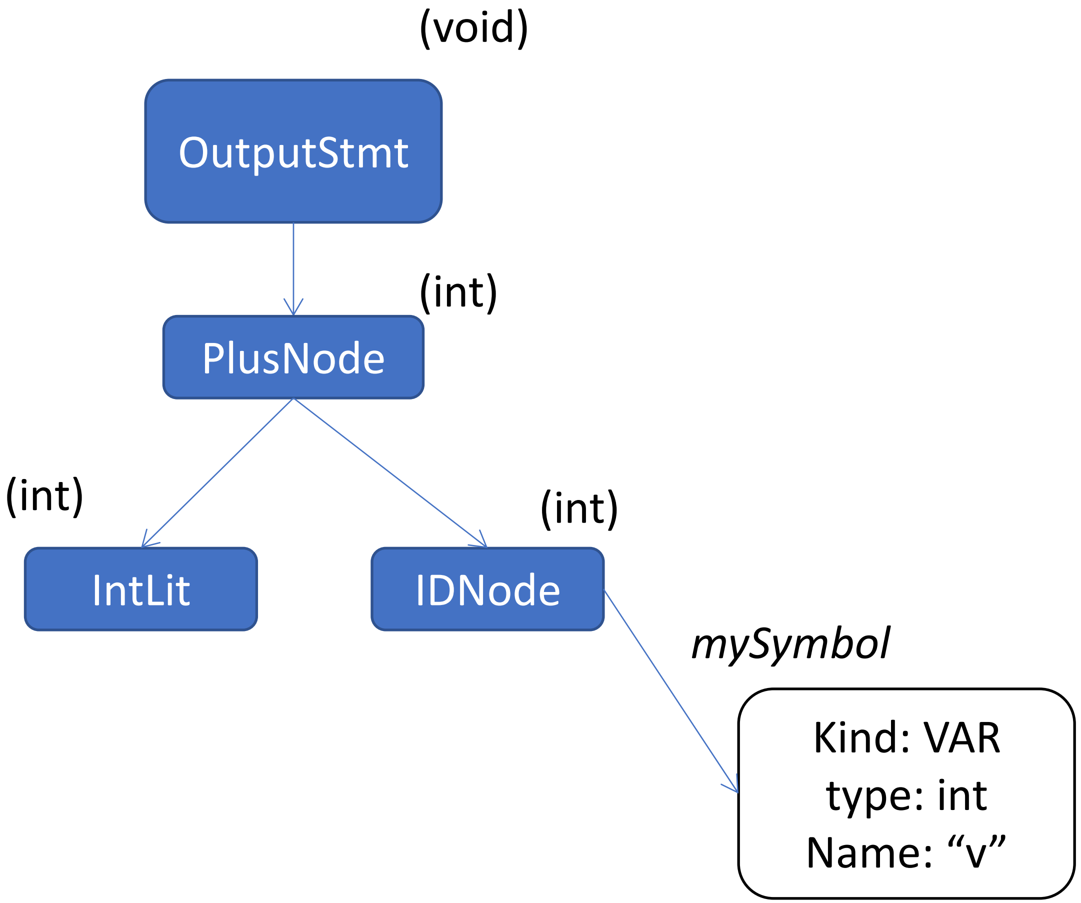
Kind: Func
Type: int,int → bool
Name: "greaterThan"

Statements

Implementing Type Checking

Always have void type

- Make sure to check child expression
- No type to propagate
- Some versions of analysis may propagate boolean: error / no error



Other AST Node Types

Implementing Type Checking

Follow these same principles

- Ensure that children are well-typed
- Apply a combination rule
  - If valid: infer a type and propagate out
  - If invalid: propagate error

while(eh?){
to console "!";
}

maybe X means 2 otherwise 3;

eh? ← bool

maybe a means 1 otherwise 2;

Exercise: Draw Type Analysis

Bonus Exercise

```
int a;
bool f;
int m(int arg){
    int b;
    return arg + 1;
}
```

Handling Errors

Implementing Type Checking

- We’d like all *distinct* errors at the same time
  - Don’t give up at the first error
  - Don’t report the same error multiple times
- When you get error as an operand
  - Don’t (re)report an error
  - Again, pass **error** up the tree

```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)


***       gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```

Operator Errors vs Operand Errors

Implementing Type Checking

The difference between…

`true + false`

error *error*

Neither operand works with the operator

… and

`true == 7`

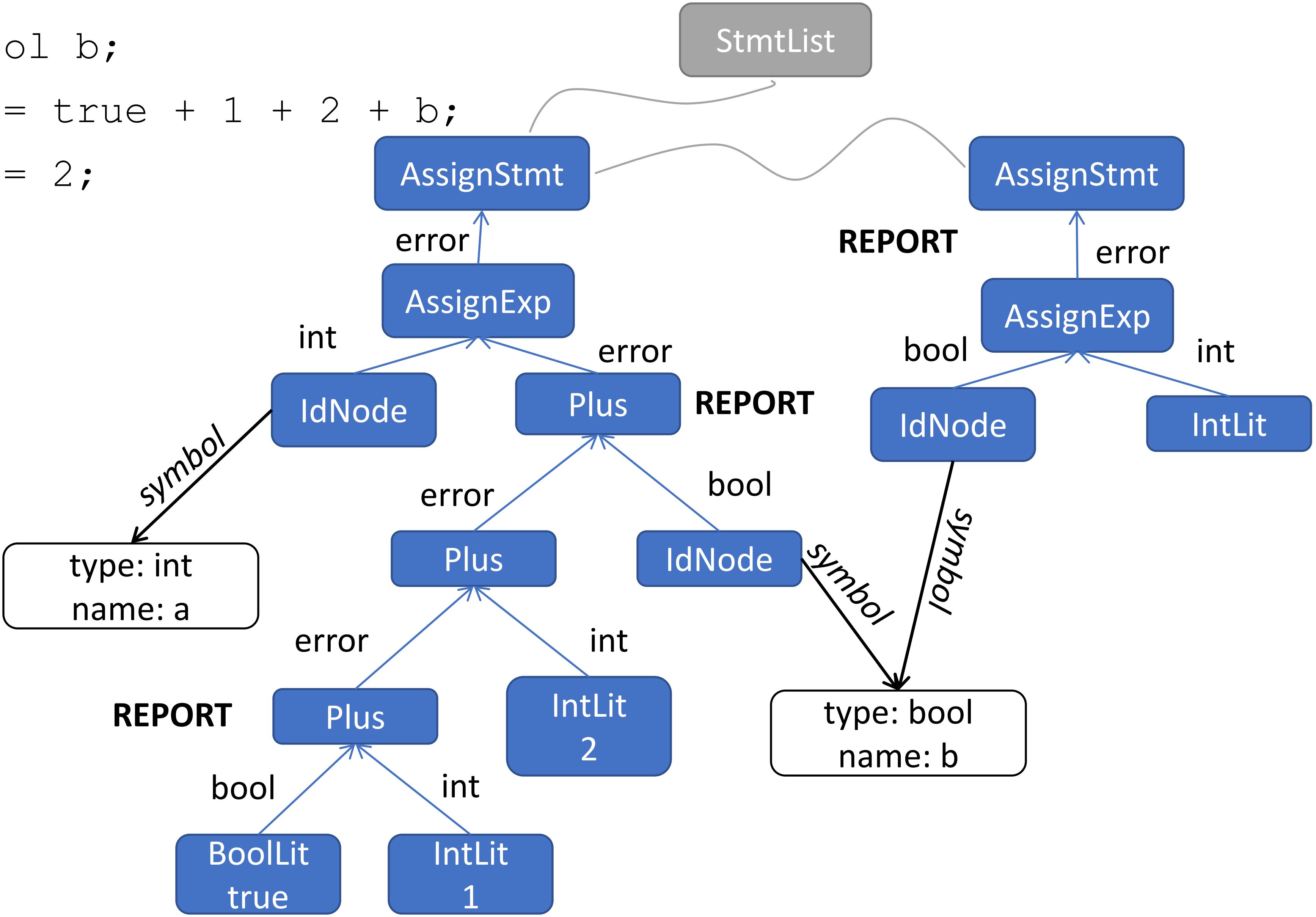
error

These operands could work with the operator
… but they don't work with each other

Type Error Example

Implementing Type Checking

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



Lecture Summary

Wrap-Up: Typechecking

- We'd like all *distinct* errors at the same time
  - Don't give up at the first error
  - Don't report the same error multiple times
- When you get error as an operand
  - Don't (re)report an error
  - Again, pass **error** up the tree

Next Time

Preview: Error Reports

Having explorer two semantic analyses, let’s generalize

- What’s the limit of semantic analysis, especially error checking?