Typechecking Intermediate and Assembly Language Programs

Lecture 20

Lecture Outline

- Why typecheck low-level code?
- Low-level typechecking difficulties
- Typechecking using global data flow analysis
- The Coolaid typechecker

Why Typecheck Low-level Code?

Reason 1: Security

- Well-typed Cool (or Java) code cannot access arbitrary memory
  - You have some safety guarantees even if you do not trust (or know) the author of the code
- It is convenient to distribute intermediate or machine code, instead of source code
  - No need to run (and trust) a compiler on the client system
  - To protect the intellectual property present in source

- The Java bytecode verifier is such a typechecker !

Why Typecheck Intermediate Code?

Reason 2: Compiler debugging

- A correct Cool compiler should never produce code that crashes (i.e., memory unsafe code)
  - The program is well-typed before code generation
  - It should remain well-typed, hence memory safe, after code generation

- A typing error in the output of a Cool compiler is always a symptom of a compiler error

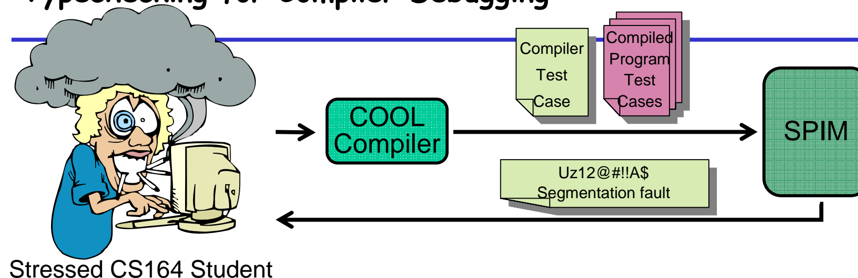
- Almost all compiler errors result in code that fails to typecheck

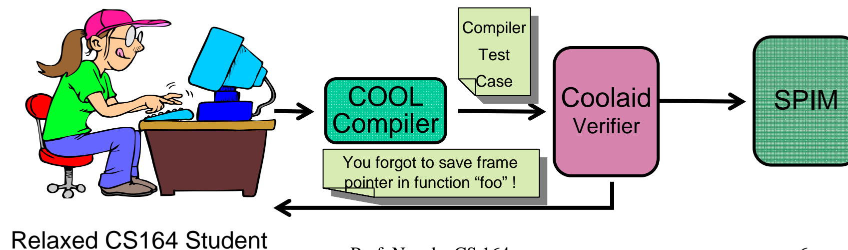
Old-Style Compiler Testing

- You must write test cases for the compiler
  - e.g., programs that you compile
- You must also write test inputs for your test cases
  - so that you can check that they run as expected

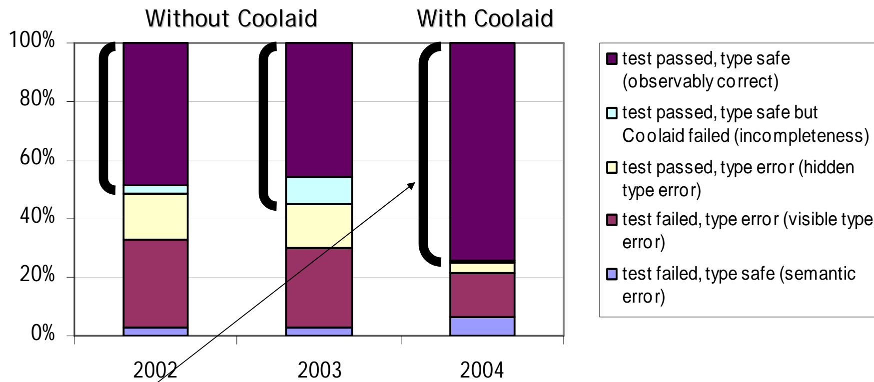
- It is very hard to exercise the entire compiler
- It is even harder to exercise completely each test program
- When a test program produces incorrect results, it is hard to identify the offending instructions

Typechecking for Compiler Debugging



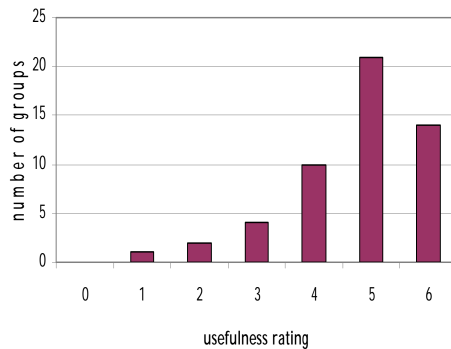


Typechecking Makes Testing More Effective



- Good compilations increase: 53% to 75%
- Testing misses 30% of compilation errors
- False alarms ratio decreases

Student Perspective (Spring 2004)



- Favorite comments:

  "I would be totally lost without Coolaid. I learn best when I am using it hands-on …. I was able to really understand stack conventions and optimizations and to appreciate them."

  "We give it a 5. We'd give it a 6 if it had an MP3 player"

0: "counter-productive"
6: "can't imagine writing a compiler without it"

Typechecking Challenges for Low-Level Code

- Variables (registers) are not used consistently with the same type
  - Because registers are reused frequently
  - The type of a register depends on the program point
- High-level operations are "unbundled"
  - a method call: null check, fetch dispatch, fetch method, save return address, call
  - allocation and initialization may be separate
- Compiler bookkeeping code is visible
  - stack allocation of temporaries
  - calling conventions

Example

- Consider two Cool classes:

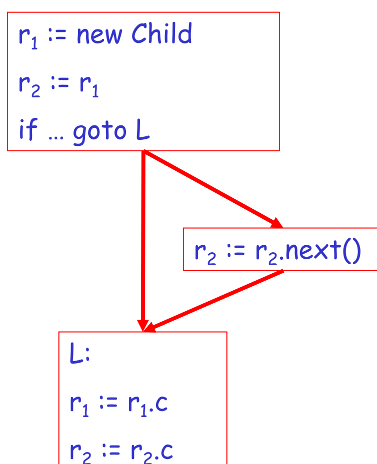
```
Class Parent {
      p : Int;
      next() : Parent { … }
}
Class Child inherits Parent {
      c : Int;
}
```

A High-Level Intermediate Language

- Consider the following intermediate language
  - Very similar to Java VML or Microsoft CIL
  - Operands are in registers, or are constants

```
I ::= r := r'
   |  r := new C          // Allocation
   |  r := r'.f           // Null-check + field read
   |  r := r'.m(r")       // Null-check + dynamic dispatch
   |  L:
   |  if r = 0 then jump L
   |  jump L
```


Flow Sensitive Type Checking



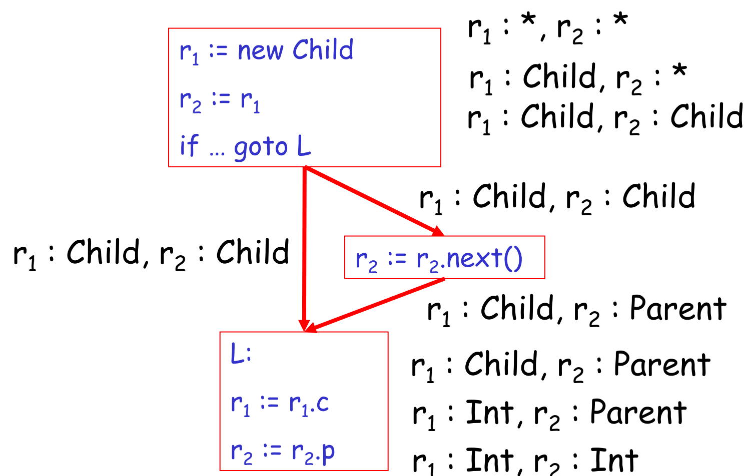
- Registers r_1 and r_2 are used with various types at different points
- We say that we need <u>flow sensitive analysis</u> of types
- Source-level type checking is <u>flow insensitive</u>
- Liveness, constant propagation analyses are flow sensitive

Type Analysis

- We will use global data-flow analysis to do type checking
- We associate one of the following values with each register at every program point

value	*interpretation*
r : #	This statement is not reachable
r : C	r contains an object of dynamic type ≤ C
r : *	Don't know the type of r

Flow Sensitive Type Checking: Example



Global Type Analysis

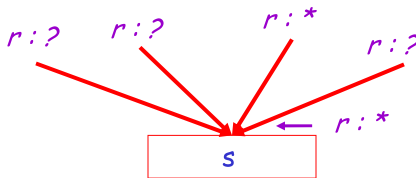
- For a statement s, compute the type of r immediately before and after s

$$T_{in}(r,s) = \text{type of } r \text{ before } s$$
$$T_{out}(r,s) = \text{type of } r \text{ after } s$$

- Define a <u>transfer function</u> that transfers type information from one statement to another

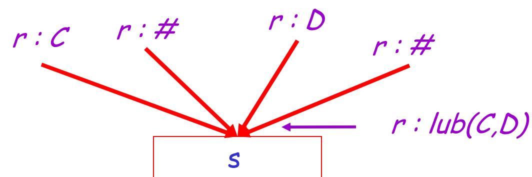
- In the following rules, let statement s have immediate predecessor statements $p_1,\dots,p_n$

Rule: Join Unknown



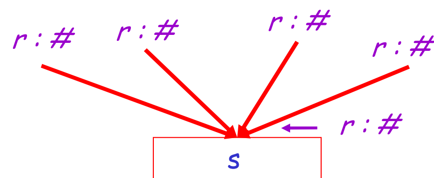
if $T_{out}(r, p_i) = *$ for some i, then $T_{in}(r, s) = *$

Rule: Join LUB



if $T_{out}(r, p_i)$ = C or D, or # for all i,
then $T_{in}(r, s)$ = lub(C, D)

Rule: Join Unreachable



if $T_{out}(r, p_i)$ = # for all i,
then $T_{in}(r, s)$ = #

Rule: Join Consolidated

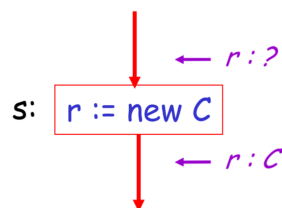
- We can consolidate all previous rules for join
- If $p_1, \ldots, p_n$ are the predecessors of s

$$T_{in}(r, s) = lub(\ T_{out}(r, p_1), \ldots, T_{out}(r, p_n))$$

We define lub also for # and *

- lub(#, T) = lub(T, #) = T
- lub(*, T) = lub(T, *) = *
- lub(C, D) = … as in the typing rules

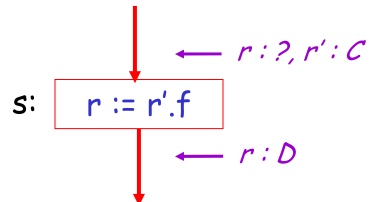
Rule: New



$T_{out}(r, s) = C$

$T_{out}(r', s) = T_{in}(r', s)$ for r' <> r

Rule: Field Access

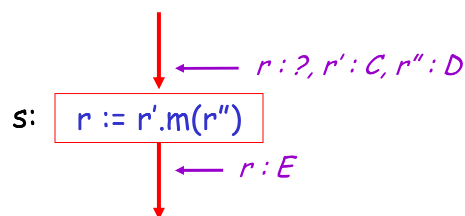


- If $T_{in}(r', s) = C$ and objects of type C have a field f of type D:

$T_{out}(r, s) = D$

$T_{out}(r'', s) = T_{in}(r'', s)$ for $r'' <> r$

Rule: Method Call



- If $T_{in}(r', s) = C$ and $T_{in}(r'', s) \leq D$, and objects of type C have a method $m(x : D) : E$

$T_{out}(r, s) = E$

$T_{out}(r''', s) = T_{in}(r''', s)$ for $r''' <> r$

Type Checking Algorithm

- For a method in class C

  m(x : D) : E

  we assume that the self argument is passed in r_{arg0} and x in register r_{arg1}

  - We start in initial type state

    r_{arg0} : C, r_{arg1} : D, r_1 : * , …, r_n : *

    All other program points are initialized with #
  - We apply the rules from before, until no more changes occur
  - If we succeed, then the program is well-typed
- Why does this algorithm terminate ?

Additional Complications

- What we described so far is Java bytecode verification, or Microsoft CIL verification
- Assembly language programs are more difficult to verify
- Because we do not have high level instructions, such as
  - r := r'.f
  - r := r'.m(r")

Handling Low-Level Field Accesses

- Consider the following object layout
- Instead of
  r := r'.f
  we have
  if r' = 0 then goto abort
  t := r' + 12
  r := mem [t]

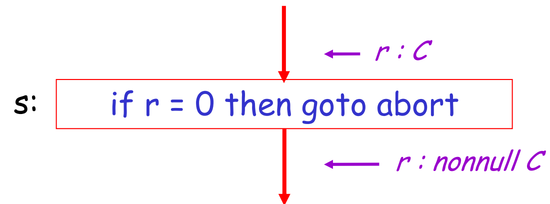
	Offset
Class Tag	*0*
Object Size	*4*
Dispatch Ptr	*8*
Attribute 1	*12*
...	*16*

- We must record that we have done the null-check
- We must give a "type" to r' + 12

More Refined Types

- We introduce the type qualifier nonnull
  - r : nonnull C means that r contains the address of an actual object of dynamic type $\leq$ C (it is not null)
  - r : C means that r may be null
- We introduce the arithmetic type nonnull C + n
  r : nonull C + n means that r contains n plus the address of an object of type C

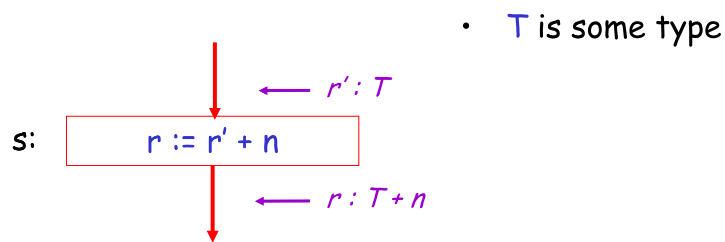
Rule: Null check



$T_{out}(r, s)$ = nonnull *C* if $T_{in}(r, s)$ = *C*

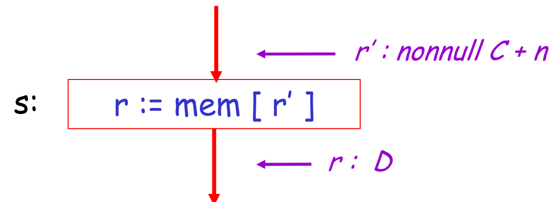
Rule: Arithmetic

- T is some type



$T_{out}(r, s) = T + n$ if $T_{in}(r', s) = T$

Rule: Low-level Field Access



$T_{out}(r, s) = D$ if

- $T_{in}(r', s)$ = nonnull $C + n$, and
- objects of class C have a field of type D at offset n

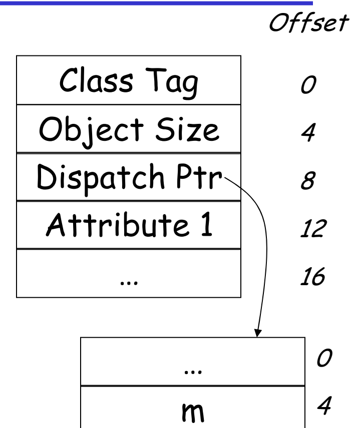
Handling Low-Level Method Calls

- Instead of

  r := r'.m(r'')

  we have

  if r' = 0 then goto abort
  r_{arg0} := r'
  t := r_{arg0} + 8
  t' := mem [t]
  t'' := mem [t' + 4]
  r_{RA} = L
  r_{arg1} := r''
  jump [t'']
  L:



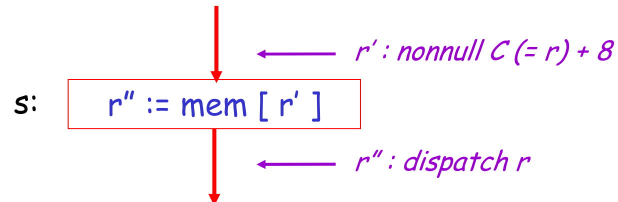
Surprising Difficulty with Method Calls

- The self parameter to a method must be the same object from which we got the dispatch table
  - Not sufficient just to have the same (static) type
  - If r has dynamic type Child and static type Parent, we will use the method of Child. We cannot use it on a Parent
- The type checker must keep track of facts such as "register t' is the dispatch table for object in register r"
- Thus, the type checker must keep track of equalities of objects, not just their types !

Refined Types for Handling Method Calls

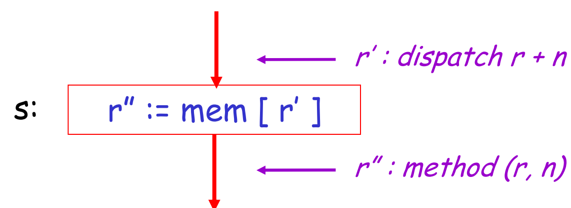
- Add new types
  - &L - the address of label L
  - dispatch r - the dispatch table of object stored in r
  - method (r, n) - the method stored at offset n in table of r
  - nonnull C (= r) – non-null object of class C, equal to object in r
- Such types are called dependent types
  - Type depends on the value of some variable
  - A very powerful form of type

Rule: Fetch Dispatch



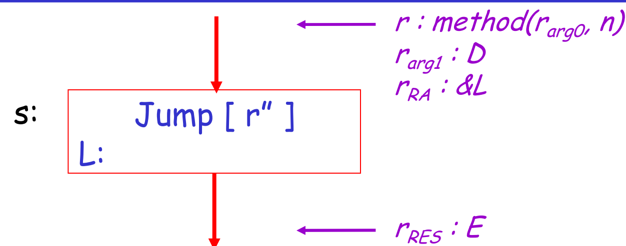
T_{out}(r", s) = dispatch r
 if T_{in}(r', s) = nonnull C (=r) + 8

Rule: Fetch Method



T_{out}(r", s) = method(r, n)
 if T_{in}(r', s) = dispatch r + n

Rule: Low-Level Method Dispatch



$T_{out}(r_{RES}, s) = E$
if $T_{in}(r, s) = method(r_{arg0}, n)$ and
$T_{in}(r_{arg0}, s) = nonnull\ C$ and
$T_{in}(r_{arg1}, s) = D$ and
objects of class C have method "m(x : D) : E" at offset n

Low-Level Method Dispatch: Example

Code	Types
	r′ : C, r″ : D
if r′ = 0 then goto abort	
	r′ : nonnull C, r″ : D
r_{arg0} := r′	r_{arg0} : nonnull C, r″ : D
t := r_{arg0} + 8	r_{arg0} : nonnull C, t : nonnull C (= r_{arg0}) + 8, r″ : D
t′ := mem [t]	r_{arg0} : nonnull C, t′ : dispatch(r_{arg0}), r″ : D
t″ := mem [t′ + 4]	r_{arg0} : nonnull C, t″ : method(r_{arg0}, 4), r″ : D
r_{RA} = L	r_{arg0} : nonnull C, t″ : method(r_{arg0}, 4), r″ : D, r_{RA} : &L
r_{arg1} := r″	r_{arg0} : nonnull C, t″ : method(r_{arg0}, 4), r″ : D, r_{RA} : &L, r_{arg1} : D
jump [t″]	
L:	

Opportunities for Optimization

- The compiler can reorder and mix instructions
  - To optimize the utilization of pipeline
  - Called instruction scheduling (we'll talk about it later)
- The compiler can even eliminate redundant computation
  - Duplicate fetching of dispatch table, or null check
  - E.g., when compiling

    r.m(1); r.m(2); t := r.f

- Our typechecker will still able to typecheck the code

Handling of Stack and Calling Conventions

The typechecker keeps track of the stack usage

- Which registers contain stack addresses:
  - SP0 + 4 means that SP on method entry plus 4
- The contents of stack slots
  - [SP0 + 4] is treated as a pseudo-register

The typechecker keeps track of return address, callee-save registers, also

- Will check that you use them correctly

Typechecking Can Miss Compiler Bugs

- We will only discover
  - Violations of the typing rules
  - Violations of the rules for manipulating the runtime data structures
- Typechecking does not compare the meaning of the output with that of the input
  - A more elaborate technique called "translation validation" does that
- Coolaid missed 80 errors this way (out of 1000 found with SPIM)
- Regular testing is still important

Conclusions

- Program analysis can be used not just for optimization but also for debugging, or security enforcement
- This is a vibrant area of research today
- You should use for your project the Coolaid tool that incorporates all of these ideas, customized for checking Cool programs

Thanks to Evan Chang, Robert Schneck, Adam Chlipala, Kun Gao