Type-safety and Compilers

CSE 501
Lecture 5

April 14, 2013
Preliminaries

- Critiques?
  - A Verifiable SSA Program Representation for Aggressive Compiler Optimization
  - Fault-safe Code Motion for Type-safe Languages
- Questions on HW 1?
Type safety

- A property of a programming language: a language is type-safe if it guarantees the integrity of its data types:
  - An int is a really an int.
  - A String is really a String.
  - A char[] is really a char[].
- Implies *memory safety*
Type-safe languages

- Unsafe languages: C, C++, ...
  - Pointer arithmetic, casting, manual deallocation can generate pointers to invalid memory.
- Safe languages: Java, C#, Python, JavaScript, ML, ...
  - Static checks: Deref. of an int won’t compile
  - Dynamic checks: Null check on a field access
  - Automatic garbage collection
Example

Java source

```java
if (a != null)
    while (!done) {
        b = (B) a;
        ... = ... b.x ...;
        ...
    }
```

Java developer
- compiles to bytecode
- ships to client

Client
- verifies bytecode
- compile to native code
- executes

Bytecode (simplified)

```java
ifnull a goto EXIT
L:
  ifeq done, 0 goto EXIT
  b = checkcast a, B
  t1 = getfield b, B::x
  ...
  goto L
EXIT:
```

Verifiably type-safe
- checkcast :
  - runtime type check
- getfield :
  - statically typed
  - runtime null check
Why type-safety?

• Security, reliability, debuggability
  • Buffer overflows, etc., common exploits
  • Untrusted code, e.g., in web browsers
  • No seg faults or mysterious corruption

• Aside: Is start IR type-safe?
How does type safety impact compilation?

- For performance, optimizing compiler must:
  - Eliminate null / type / bounds checks when possible
  - Preserve the safety guarantees of the language - cannot just elide all checks!
  - Preserve effectiveness of classical optimizations
  - Better yet, do it all in limited compilation time
It’s not all bad....

• Types are very useful in compilation
• Debug optimizations / validate generated code
• Enable optimizations - e.g., redundancy elimination

```c
void foo(A a, B b) {
    ...
    t1 = b.x;
    a.x = ...;
    t2 = b.x;  <- Is this redundant?
    ...
```

• What if A and B are unrelated classes?
• Yes : type-based alias analysis (Diwan, ...)

Tuesday, April 23, 13
It’s not all bad....

- Types are very useful in compilation
  - E.g., Java to bytecode compilation
  - Preserves safety information
  - Untrusted Java bytecodes can run safely in a browser
- Ideally, keep type info as long as possible (native?)
  - Debug optimizations / validate generated code
  - Ship/store/cache optimized, verifiable code
Background: StarJIT

- Intel invested heavily in Itanium architecture
- Pushed complexity from hardware to the compiler: e.g., instruction scheduling, parallelism
- Relied heavily on compilers for performance
- Concerned that Java JIT wasn’t up to the task
- Goal: demonstrate high-perf Java on Itanium ... or prove it cannot be done
StarJIT

- Java / .NET JIT for Itanium and x86
  - High performance (best Java on Itanium)
  - SSA low-level IR with type information
  - Java-specific opts for check elimination, inlining
  - Aggressive “classical” optimizations
  - Exploit types to drive optimization
  - Preserve type info as long possible
- Part of Apache Harmony’s Java implementation
Bytecodes impede optimization

Bytecode

```java
ifnull a goto EXIT
L:
ifeq done, 0 goto EXIT
b = checkcast a, B
t1 = getfield b, B::x
...
goto L
EXIT:
```

Redundancies: 3 null tests
- ifnull
- checkcast
- getfield

Aliasing
- a == b
- breaks opts (e.g., PRE) that depend on syntactic equivalence
Erasure-style IR

Bytecode

```java
ifnull a goto EXIT
L:
    ifeq done, 0 goto EXIT
    b = checkcast a, B
    t1 = getfield b, B::x
    ...
    goto L
EXIT:
```

Lowered IR

```java
if a == null goto EXIT
L:
    if done == 0 goto EXIT
    checkcast a, B
    checknull a
    t2 = getfieldaddr a, B::x
    t1 = load [t2]
    ...
    goto L
EXIT:
```

In Java
- checks may throw exception
- load *cannot* fault
Erasure-style IR

Lowered IR

```java
if a == null goto EXIT
L:
  if done == 0 goto EXIT
  checkcast a, B
  checknull a
  t2 = getfieldaddr a, B::x
  t1 = load [t2]
  ...
  goto L
EXIT:
```

Optimized IR

```java
if a == null goto EXIT
if done == 0 goto EXIT
checkcast a, B

L:
  t2 = getfieldaddr a, B::x
  if a == null goto EXIT
  if done == 0 goto EXIT
  checkcast a, B
  checknull a
  t2 = getfieldaddr a, B::x
  t1 = load [t2]
  ...
  goto L
EXIT:
```

Safety is obscured....

Optimizations:
- CSE, partial peeling, hoisting
Refinement-style IR

Lowered IR

```java
if a == null goto EXIT
L:
  if done == 0 goto EXIT
  b = checkcast a, B
  t3 = checknull b
  t2 = getfieldaddr t3, B::x
  t1 = load [t2]
  ...
goto L
EXIT:
```

The good
- Type refinement helps preserve safety
  - b is of type B
  - t3 is nonnull(B)
  - t2 is a safe pointer

The bad
- Artificial constraints on code motion
- Naming: redundant null checks now obscured
Proof passing IR

Lowered IR

```java
[_,pf1]if a == null goto EXIT
L:
    if done == 0 goto EXIT
    pf2 = checkcast a, B
    pf3 = checknull a
    t2 = getfieldaddr a, B::x
    pf4 = pfand(pf2, pf3)
    t1 = load [t2] (pf4)
    ...
    goto L
EXIT:
```

Key idea:
- factor out safety
- represent explicitly
- abstract proof variables
- load req. safety proof

Benefits:
- artificial constraints gone
- no duplicate names
- safety just a value dependence
Lowered IR

```java
[_,pf1]if a == null goto EXIT
L:
  if done == 0 goto EXIT
  pf2 = checkcast a, B
  pf3 = checknull a
  t2 = getfieldaddr a, B::x
  pf4 = pfand(pf2, pf3)
  t1 = load [t2] (pf4)
  ...
goto L
EXIT:
```

Optimized IR

```java
t2 = getfieldaddr a, B::x
[_,pf1]if a == null goto EXIT
  if done == 0 goto EXIT
  pf2 = checkcast a, B
  pf4 = pfand(pf2, pf1)
L:
  t1 = load [t2] (pf4)
  ...
  if done != 0 goto L
EXIT:
```

Optimized and verifiable.
Verification

- \( t_2 = \text{ldind } t_1 (pfl) \)
  - \( t_2 \) has type \( T_1 \)
  - \( t_1 \) has type \( *T_2 \) where \( T_2 \) is a subtype of \( T_1 \)
  - \( pfl \) implies that \( t_1 \) is non-null and in-bounds
  - decidable via Presburger arithmetic
SSA

- Proofs are SSA along with everything else
- \( pf3 = \phi(pf1, pf2) \)
  - \( pf1 \rightarrow pf3 \) and \( pf2 \rightarrow pf3 \)

- Modular verification
  - If all instructions type check, program type-checks
Safe speculative code motion

• Speculative code motion

• Insert potentially unsafe operations onto new paths if profitable

• Prior techniques: use recovery mechanism (e.g., Itanium ld.s / chk.s)

• New approach: exploit type safety in IR

• Applicable to: loads, divides, ....
Example

```java
k = 0;
if(a != null) {
    for(int i=0; i<n; ++i){
        if((i & mask) != 0)
            k += a.x/3;
    }
}
```

Want to hoist `a.x/3` out of loop
- E.g., profile says expr. is hot
- but, load may fault
- cannot insert into new path
- PRE, etc., cannot optimize
- Illegal to hoist in C/C++
- In Java?
```java
k = 0;
if(a != null) {
    for(int i=0; i<n; ++i){
        if((i & mask) != 0)
            k += a.x/3;
    }
}
```

Down-safe
k = 0;

if(a != null) {
    for(int i=0; i<n; ++i){
        if((i & mask) != 0)
            k += a.x/3;
    }
}
k = 0;
if(a != null) {
    for(int i=0; i<n; ++i){
        if((i & mask) != 0)
            k += a.x/3;
    }
}
Fault safety

```java
k = 0;

[pf1, _] if (a != null) {
    for (int i = 0; i < n; ++i) {
        if ((i & mask) != 0)
            k += a.x [pf1]/3;
    }
}
```

Fault-safe!
k = 0;

pf1, _ if (a != null) {
    tmp = a.x [pf1] / 3;
    for (int i = 0; i < n; ++i) {
        if ((i & mask) != 0)
            k += tmp;
    }
}

load a.x (pf1)
Implementation

- Untyped proof variables (preserved through opts)
- Two code motion optimizations
  - Fault-safe Speculative SSA-PRE
  - Fault-safe Itanium instruction scheduling
SpecJVM Performance

Normalized Execution Time on 1.5 GHz Itanium 2

- _201_compress
- _222_mpegaudio
- _227_mtrt

- Base
- FS SSA-PRE
- FS IPF IS
- FS All
Further reading

• Proof carrying code for Java (Colby, et al, PLDI 00)
• Typed assembly language (Morrisett, ....)
• Type-preserving compilation (Chen, et al, PLDI 08)
• Type-based alias analysis (Diwan, et al, PLDI 98)