CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 37: Exceptional topics

Compiler project

- Due date: December 16
  - –Accepted (late) until December 18. **Hard deadline.**
- No room for error—plan early and often
  - –Got test cases?
- Cool Qt-based UI library coming soon...
- Compiler competition!
  - –Correctness, speed, compiler engineering
  - –Winners receive plaque, bragging rights.

Exceptions

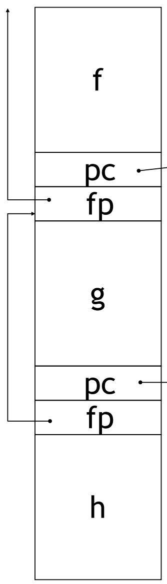
- Many languages allow *exceptions:* alternate return paths from a function
  - – null pointer, overflow, emptyStack,...
- Function either terminates *normally* or with an exception
  - – *total* functions ⇒ robust software
  - – normal case code separated from unusual cases
  - – no ignorable encoding of error conditions in result (e.g., null)
- Exception propagates *dynamically* to nearest enclosing try..catch statement (up call tree)
  - -Tricky to implement dynamic exceptions efficiently
  - -Result: underused by programmers (see Map.get, etc.)

Exceptions: goals

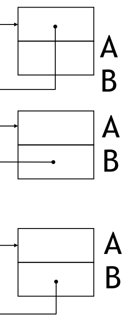
1. normal return w little or no added overhead
2. try/catch free if no exception
3. catching exception ~ cheap as checking for error value
   - –C/C++: setjmp/longjmp. Try/catch expensive.

- **Static exception tables** (CLU):
  - –insight: can map pc to handler w/in each function.
  - –on exception: climb stack using return pc, look up exception handler at each stack frame (binary search on pc)

Example




```
f() throws B = (
   try g()
      catch A => S1
   try h()
      catch B => S2
)
g() throws A, B = (
   try h()
      catch B => S3
)
h() throws A,B = (
   throw A
)
```



– need to store callee-save regs on walk

– lookup complicated if exc names = classes – need dispatch

Run-time type discrimination

- How to discover types at run time?
  - n tag bits ⇒Tag 2^n-1 primitives, align memory to 2^{n-2} words, some performance hit, range limitation on ints (x→ 2^n x)
- o instanceof T, (T)o, typecase o of T_1⇒ s_1 | T_2⇒ s_2
  1. look up DT pointer, class descriptor in hash table containing type relationships (may be filled lazily)
  2. (SI only, separate compilation) Record superclasses sequentially in DT (**display**). instanceof C ⇒ check if class at depth depth(C) is C.
  3. (Single inheritance only) in-order traversal of hierarchy with classes numbered sequentially ⇒ all subclasses of C in contiguous range. Test class index in range with single unsigned comparison.
  4. Quick range test (ala #2) can be done even with MI using **PQ-trees**.

Metaobjects

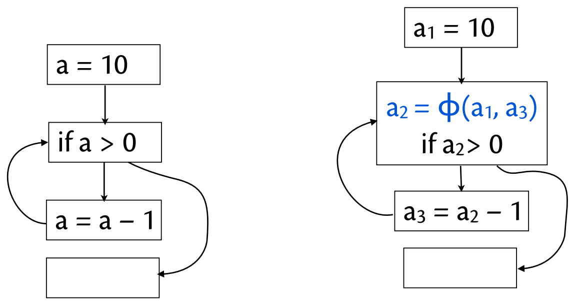
- Some languages (Smalltalk, Java, ...) expose classes as objects (metaobjects)
  - query methods, fields, inheritance structure...
  - good for building compilers, run-time adapters, serialization code... *not regular code*
- Metaobject protocol: methods exposed for querying classes, other type-level entities
- Java 1.5⁺: parametric polymorphism not reflected – really *JVM* metaobjects

Generalized LR parsing

- Some parser generators (e.g. PPG) support **grammar inheritance** to support language extension
  - Problem: LALR grammars are not very extensible
- GLR parsing: conflicts resolved late by forking the parser stack. Compiler must reconcile alternate parsing results.
- Another nice idea: parser feedback to lexer to identify next legal tokens

Static Single Assignment (SSA)

- Intermediate language form: every variable has exactly one definition
  - variables are immutable ⇒ simplified analyses and code transformations
  - close correspondence to functional style (see Appel)
  - Need extra "phi" nodes indexed by incoming edge
- Extra dataflow analyses needed for conversion to SSA.



Combine info about a_1, a_3 into a_2

Path and object sensitivity

- **Flow-insensitive:** same information throughout code (type checking)
- **Flow-sensitive:** information per program point
- **Context-sensitive:** information per calling context
- **Path-sensitive:** information per execution path leading to program point.
- **Object-sensitive:** information per method receiver object. Helps with points-to analysis.

Speeding up dataflow analysis

- Expensive to rerun analysis after each optimization!
- Incremental analysis: "fix up" analysis results to deal with optimizations.
- Cascading analysis: build expected optimization into the analysis.
- Composition of analyses also possible (Vortex compiler)

Abstract interpretation

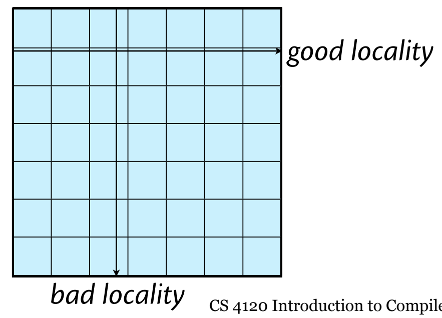
- Many forward analyses can be viewed as instances of **abstract interpretation**
- Idea: analysis ~ running the program, but mapping actual program state to a simplified abstract state.
  - Example: points-to analysis using abstract heap, a relation on "variables" and "objects".
- Transfer function is an abstraction of computation. Maps input abstraction to an output abstraction that includes all feasible concrete outputs.
- Convergence = run loops until abstract state converges.
- A rich mathematical structure for explaining and developing program analysis.

Attribute grammars

- Essentially a type system for program analysis and synthesis, with extra constraints in rules.
- Typing rules generate additional information about program (analysis results, output machine code, ...)
- Iterative constraint solving, not recursive type checking – information flows up and down in AST in complex ways.
- Examples:
  - Synthesizer Generator (Teitelbaum): a Cornell compiler framework based on attribute grammars.
  - JastAdd: a Java compiler based on attribute grammars.

Optimizing for locality

- 100^+-fold speed difference between memory and cache ⇒ locality is crucial for performance.
- Inlining objects and arrays into referencing structures avoids indirection, requires exact type and escape analysis.
- Some important tricks for matrices:



1. Transpose matrices so loops go across rows.
2. Pad rows to avoid cache conflicts
3. Rewrite nested loops with outer loops over **blocks,** inner loop within each block.

Instruction scheduling

- Key: want to keep every pipeline stage of processor busy.
- Order of instructions matters; hard to predict effect.
  - Start load instructions early
  - Intel: compiles instructions to RISC-like **micro-ops**.
- Instruction scheduling: low-level optimization on assembly code.
  - Reorder instructions subject to dependencies between instructions (topological sort, need alias analysis...)
  - Scheduling is traversing dependency DAG on instructions
    - heuristics to start important work early, keep functional units busy.
  - Knowing ISA is not enough.
- Need to schedule before *and* after register allocation.

Type-preserving compilation

- Idea: compiler propagates types to compiled code. **Verifier** checks to see compiled code is safe.
  - Code consumer doesn't have to trust compiler or compiled code.
  - Examples: Java bytecode, Typed Assembly Language (TAL).
- Bytecode verification is a dataflow analysis.
  - Dataflow values = mapping from locals, stack locations to types.
- Challenge: low-level code needs complex types. (Type of stack pointer? program counter?)

Closing thoughts

- Being able to build a compiler opens new opportunities for solving problems. Valuable knowledge!
  - Many uses for domain-specific languages—look for opportunities to use them.
  - C, Java are pretty good target languages – let someone else write the optimizer (except: exceptions, threads, coroutines, dispatching, transactions, …)
- Possible next steps:
  - CS 6110: Advanced programming languages (theory, SP10)
  - CS 4110: Programming languages (features, FA10)
  - CS 6120: Advanced compilers, not offered soon.
  - TA this course in FA11