





COMP 181

Lecture 20 The Last Lecture

December 7, 2005


Prelude

- What is the Hubble Constant?
 - From cosmology, rate of recession of nearby stars, galaxies
- Based on observation
 - At this scale, structures move apart at a speed proportional to their distance.
- Deceleration parameter q
 - Long thought to be positive – Universe expansion slowing
 - 1998: Apparently, q is negative – expansion is accelerating
- The end
 - Heat death?
 - Dark energy rips apart the Universe – the Big Rip
 - Dark energy becomes attractive – the Big Crunch

Tufts University Computer Science 2

Today




- A bit more on functional languages

Wrap up:

- Things we didn't talk about
- Future directions
- So, you want to compiler research...
- So, you want a job working on compilers...

Tufts University Computer Science 3


Functional languages



- Last time
 - Lambda calculus
 - Interpreters
 - Compiling to an abstract machine
 - Compiling to machine code
- Key: closures
 - Shared code
 - Free variables in environment
- Data structures
 - Built up from functions
 - In practice: data structure (e.g., number, lists) provided as primitives

Tufts University Computer Science 4


Data representations



- Data structures tend to be high-level
 - Example: Lists of lists
 - Few explicit constraints on implementation
 - Very flexible types
- Double-edged sword – why?
 - Leaves room for the compiler to do something clever
 - Puts the burden on the compiler to do something clever

Tufts University Computer Science 5

Without static typing



- Scheme (Lisp)


```
(cons 5 2)
(cons (cons 5 2) "hello")
```
- Need tags to keep track of run-time types
- All data types must fit in a common format (usually a word)
 - Requires **boxing**: allocate the data on the heap, store a pointer in the word
 - Floats are usually boxed – they require 8 bytes
 - Records are boxed
 - Arrays are often arrays of pointers to boxed elements
- You can see the performance issue

Tufts University Computer Science 6

With static typing

C, Pascal

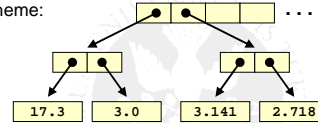
- Types are known at compile time
- No need for run-time type tests (mostly)
- Different data types can have different sizes
 - Unboxed floats
 - Even unboxed records (if small enough)
 - Arrays of primitive types (flat)
- Note: assumes functions are *monomorphic*
 - Only defined for one set of input types – one signature
 - Allows the compiler to use type-specific calling conventions
 - Example: pass float arguments in FP registers



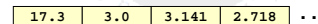
Example

- Array of points in 2D space

- In Scheme:



- In C:



Polymorphic typing

- Functions can accept arguments with different type signatures
 - Very natural in functional languages
 - Limited form in C++ and Java
- The compiler (type system) guarantees type safety
- However:
 - It doesn't assign unique types to variables at compile time
 - Some variables may have unknown types
 - Therefore: unknown sizes
 - All associated problems



Possible solutions

- Restrict polymorphism to "record" types
 - Modula: abstract types must be pointer types
 - Java: types "int" and "float" cannot be coerced to Object
 - Code replication
 - Ada, C++: compile a specialized version of a generic function for each potential type signature
 - Revert to Scheme-style data representation
- Problem:**
- Less natural – this is a hassle in java
- Problem:**
- Possibility of code explosion
 - May require link-time code generation (C++)
- Problem:**
- Inefficient: lots of boxing and unboxing



Better solutions

- Use run-time type inspection
 - Pass type information explicitly in polymorphic code.
 - Use this information to determine sizes and layouts
- Mix different representations
 - C-style for monomorphic code
 - Scheme-style for polymorphic code
- Optimize Scheme-style representations
 - Apply local unboxing as an optimization
 - Use inlining to improve opportunities
 - Have special treatment for arrays



Type passing approach

- **Idea:**
Pass type information explicitly as extra arguments to each polymorphic function

```
let f x = x           let f α x = x
let g x = f(x, x)    let g β x = f <βxβ>(x, x)
g 5                  g <int> 5
```

- Notice: this allows f to know at run-time that its parameter is of type <int x int>
- Used in TIL – typed intermediate language
 - Use C-style "flat" data representation
 - In polymorphic code, compute sizes on the fly
 - Monomorphic code same as C approach



Example

Original code:

```
let assign_array a b i = b.(i) <- a.(i)
```

Generated code:

- Scheme-style:


```
assign_array(a, b, i) {
  load one word from a+i*4
  store in b+i*4
}
```

- TIL style:


```
assign_array(a, a, b, i) {
  s = sizeof(a);
  bcopy(a+i*s, b+i*s, s);
}
```



Mixed representations

- Use C-style representations for data whose exact type is known at compile-time
- Revert to Scheme-style representations for data whose type is not completely known at compile-time
- Insert coercions between the two at the interfaces
- Used in SML/NJ

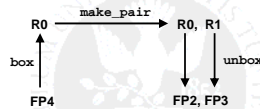


Example

Original code:

```
let make_pair x = (x, x) in ... make_pair 3.141
```

Coercions:



Generated code:

```
let make_pair x = (x, x) in
let (fst, snd) = make_pair(box(3.141)) in
(unbox(fst), unbox(snd))
```



Unboxing techniques

Start with Scheme-style boxing by default

- Perform dataflow analysis to find unboxing opportunities

```
let x = box(f) in
... unbox(x) ... unbox(x)
```

```
let x = f in
... x ... x
```

- Extend to inter-function analysis
 - Inline to expose more opportunities
 - Use control-flow analysis to determine call graph
- Use analysis to identify important cases, like array of float



Partial inlining

Called the worker-wrapper technique

- Split a function into two parts
 - A *worker function* that performs the main computation entirely on unboxed values
 - A *wrapper function* that performs the boxing and unboxing around the worker
- Goal:
 - At a call site, try to inline the wrapper function
 - Hope that its boxing and unboxing cancel out that of the caller



Example

- Worker:


```
let worker_f a b =
  (* a and b are unboxed *)
  compute on a and b
  (* return value unboxed *)
```

- Wrapper:


```
let f a b =
  box( worker_f( unbox(a) unbox(b) ))
```

- In use:


```
... unbox( f (box(3.1) box(2.7)) ) ...
```

- Optimized:


```
... worker_f 3.1 2.7 ...
```



Conclusions

- From an *engineering* standpoint
 - On comparable programs, these techniques get within 50% of optimizing C compilers
 - On some programs (allocation intensive), may run faster!
 - Getting rid of the last 50% is difficult
 - Still requires programmers to be aware of performance issues
- From a *research* standpoint
 - Promotes software reliability
 - clean semantics => formal methods => reliable programs
 - But, correct code useless if compiler is incorrect
 - These techniques allow certified compilers, and certifying compilers
 - Works for realistic bytecode compilers
 - Towards verification of optimizing native code compiler



Wrap up

Big principles

- Scanning and parsing
 - Solid theory drives most efficient implementations
 - Solved problem?
- Semantic analysis and lowering
 - Some theory – much more solid in functional languages
- Optimization
 - Analysis developed from lattice theory
 - Transformation are more ad hoc
- Code generation
 - Pattern matching often used in practice

Middle of the compiler is still a black art



Other topics

Things we didn't talk about

- Instruction scheduling
 - Move instructions around to hide memory latency
 - Highly dependent on architecture
 - Basic block scheduling
- Run-time systems
 - Memory management and garbage collections
 - There is a role for the compiler
 - Run-time representation of types
 - Relevant even in C++ and Java



Other topics

- The litany of optimizations
 - Redundancy elimination
 - Code motion
 - Loop optimizations
 - Unrolling, splitting, peeling, fusing
 - Tiling
- Memory optimizations
 - Some overlap with loop optimizations
 - Encouraging locality
- Inlining
 - Policies and heuristics



Other topics

- Dynamic optimization
 - Balancing compilation cost with run-time
 - Optimization with limited resources
 - Optimization with run-time information
- Optimizing for power/energy
 - Generating code for low power
 - Explicit management of power resources
- Compilation for correctness
 - Compiler as a code checking tool
 - Finding bugs
 - Checking for security vulnerabilities



Other topics

- Linking and loading
 - Particularly, dynamic linking
- Compiling for parallel machines
 - Different abstractions – representing parallelism
 - Distributed data structures
 - Example: HPF – High-performance Fortran
 - Automatic parallelization – dead?
- Ordering and interaction between compiler passes
 - Still a black art
 - Some work using machine learning



Future directions

Driving forces

- Architectures are getting more complicated
 - Exposing more internals in the ISA
 - Relying more on compilers
 - Examples: CELL, Itanium
- Maximum performance not always the top concern
 - Energy is a huge problem
 - More focus on reliability
 - Other metrics, like real-time
- Diversity of architectures
 - Not on the desktop
 - Servers
 - Most importantly: embedded systems



Compiler research

- Lots of great problems to work on
- Places
 - Top 4
 - Berkeley, CMU: theoretical programming languages
 - MIT: theory and systems
 - Stanford: error checking
 - Rice, UIUC
 - University of Texas (TRIPS project)
 - Rutgers, UMass, UVA, UC San Diego, U of Toronto, U of Rochester, U of Utah, Purdue, many others...
 - Tufts!



Compiler jobs

- Different levels
 - Ph.D. level: research and development
 - B.A. level: more product development
- Either way, you can work on real products
- Places
 - Intel (Oregon and Santa Clara)
 - Sun – here in Burlington, and at SunLabs in CA
 - IBM Toronto – new product JIT group
 - Microsoft
 - Small companies
 - Specialized compiler applications
 - Embedded systems



Now what?

- Take home final exam – due on Dec 16th (Friday)
 - Good news: it's a lot better than the midterm
 - Bad news: it isn't ready (haven't heard back from Noah)
- Have a good winter break
- Thanks!

