Simple Dynamic Compilation with GOO

Jonathan Bachrach
MIT AI Lab
GOO Talk

• Preliminary work
• Introduce challenge
• Present GOO
  – Introduce language
  – Sketch implementation
  – Report status
• Request Quick C– features
Scripting Languages Can Be: Fast, Clean, Safe, Powerful, and of course Fun

- Don’t have to trade off fun for speed etc.
- Don’t need complicated implementation

- Requires forced rethinking and reevaluation of
  - Technologies - faster, bigger, cheaper
  - Architecture - dynamic compilation
  - Assumptions - …
GOO

Art / Science / Education

• Research/Teaching vehicle
  – For rethinking language design and implementation

• Reaction to a Reaction …

• Targetted for high-performance/interactive software development for
  – Embedded systems
  – Electronic music
GOO Power

Features

• Pure object-oriented
• Multimethods
  – Slot accessors as well
• Dynamic types
  – Ext. param types
• Modules
• Macros
• Restartable exceptions

Builds on

• Proto
• Dylan
• Cecil
• CLOS
• Scheme
• Smalltalk
GOO Simplicity

• PLDI Core Wars
  – 10K Lines Implementation *
  – 10 Page Manual **
  – Hard Limit – “pressure makes pearls”

• Interpreter Semantics
• Speed through “partial evaluation”
• Implementation Serendipity
Complexity is Dangerous to Your Health

• Complexity will bite you at every turn

Minimize number of moving parts

• Complexity can be exponential in part count unless extreme vigilance is applied

• But vigilance is costly especially reactive

Apply vigilance towards minimizing part count instead
Simplified Design

Simplification
- No sealing
- Dynamic typing
- No databases
- Type-based opts only
- No static modeling
- Prefix syntax
- No VM

Recover x with
- Global info / d-comp
- Type inference
- Save-image
- C (C--) backend
- Use real world
- Short + nesting ops
- (Obfuscated) Source
Goal: To Develop Simple, Powerful, and Useful Techniques

- Motivated from Lightweight Languages conference at MIT 2001
- Understandable
- Adoptable
- Leveragable
GOO: Speed and Interactivity

Always optimized
Always malleable
Related Work

- Lisp Machine Progress Report, 1977, MIT
- Harlequin and Apple Dylan, 1990, Moon et al
- Adaptive Optimization For Self: Reconciling High Performance With Exploratory Programming (1994), Urs Holzle
- Java optimization in the face of class loading, 2001, ???

- Specialized hardware
- Reduced interactivity
- Increased complexity
- Reduced interactivity
Incremental Global Optimization

- Always compiled
- Dependency tracks assumptions during compilation
- Reoptimizes dependents upon change
- Knob for adjusting number of dependents to make recompilation times acceptable
Managing Complexity

1. Dynamic compilation
2. Dependency Tracking
3. Type-based optimizations
4. Subclass? tests
5. Multimethod dispatch
Complexity Example One: Dynamic Compilation

- So you want a dynamic compiler?
  - Throw away interpreter
  - Allow for more global optimizations
- But what about the ensuing complexity?
  - Use source instead of VM
    - Cut out the middle man
  - Use C back-end and shared libraries (e.g., MathMap)
    - More realistically C--
  - Trigger compiler
    - By global optimization dependencies
    - Not profile information
Using C for Simple Dynamic Compilation

• Procedure
  – Emit C code with g2c
  – Compile C code with gcc
  – Dynamically link with ld
  – Load into running image with dlopen
  – Run top level initialization code with dlvar and apply
  – Lazily resolve variables within running image

• Fast Turnaround
  – Typical interactive definitions take less than a second
Complexity Example Two: Dependency Tracking

• Assumptions
  – All optimization information is derived from bindings

• While compiling definition
  – Establish current dependent
  – Log binding accesses

• Trigger selective recompilation when
  • Dependent binding properties change

• Can decrease recompilation by
  • Recording compilation stage
  • Rerunning recorded stage and beyond
Complexity Example Three: Type-based Optimizations

- First compile loosely
  - Don’t look at binding values
- Execute resulting changes on image
  - Building objects
- Recompile with optimizations
  - Consult actual world for object properties
  - Log dependencies
Complexity Example Four: Fast Subclass? Tests

• Crucial for the performance of languages
  – Especially languages with dynamic typing
• Used for
  – typechecks
  – downcasts
  – typecase
  – method selection
• Used in
  – Compiler - static analysis
  – Runtime
Important Subclass? Measures

• The predicate speed
• The subclass data initialization speed
• The space of subclass data
• The cost of incremental changes
  – Could be full reinitialization if fast enough
Longstanding Problem

• Choose either
  – Simple algorithm with $O(n^2)$ space or
  – Complicated slower to initialize algorithm with better space properties:
    • PE – Vitek, Horspool, and Krall OOPSLA-97
    • PQE – Zibin and Gil OOPSLA-01
PVE Algorithm

• Blindingly fast to construct
  – Fast enough for incremental changes
• One page of code to implement
• Comparable to PE on wide range of real-world hierarchies
  – E.g. 95% compression on 5500 class flavors hierarchy (4MB bit matrix)
• Exhibits approximately n log n space
• Paper available: www.jbot.org/pve
Complexity Example Five: Dispatch

• For a given generic function and arguments choose the most applicable method

• Example:
  – Gen: \((+ \ x \ y)\)
  – Mets: num+ int+ flo+
  – Args: 1 2
  – Met: int+

• Typical solution is method cache
  – Concrete argument classes are keys
Subtype? Based Dispatch Methodology

Steps
- Dynamic subtype? based decision tree
  - Choose more specific specializers first
  - Choose unrelated specializers with stats
- Inline small methods
- Inline decision trees into call-sites

Examples
- \( (\text{fun } (x \ y) \ (\text{if } (\text{isa? } x <\text{int}>) \ ...))) \)
  - Discriminate int+ and flo+ before num+
  - Discriminate int+ before flo+
- int+ (and slot accessors)
- (+ x 1) (allowing partial evaluation at call-site)
Subtype? Based Dispatch
Happy Synergies

- Few moving parts
- “tag-checked” arithmetic for free
- Static dispatch for free
- One arg case comparable to vtable speed
  - Fewer indirect jumps
- Dynamic type-check insensitive to class numbering
GOO Status

**Working**
- Fully bootstrapped
- Linux and Win32 Ports
- Runtime system tuned
- C based dynamic compiler
- SWIG backend + GTK

**In progress**
- Decision tree generation
- Dependency tracking
- Fast subclass?
- Type inference
- Parameterized types
- GUI
Challenges

- Live update of objects after class redefinition
- Patching of pending functions
- Incremental interprocedural analysis
- Smart inlining
GOO Credits Etc

• Thanks to
  – Craig Chambers
  – Eric Kidd, James Knight, and Andrew Sutherland
  – Greg Sullivan
  – Howie Shrobe (and DARPA) for funding

• To be open sourced in the coming weeks:
  – www.jbot.org/goo/
Quick C-- Requests

- Dynamic Compilation
- Debugging
- GC
- Profiling
Dynamic Compilation Support

- In memory code generator
- In memory linker
- Relocatable code
- Integration with gc
C-- Debugging Support

- Source locations
- Stack walking
- Live local variables
- Execute within a frame
- Switch threads
- Force threads to safe points
GC Support

- Precise GC
- Find all references for live patching
Profiling Support

- Low overhead
- Reasonably precise
More Information

• Dynamic Languages Group
  - www.ai.mit.edu/projects/dynlangs
  - 08FEB02: MAST: A dynamic language for active network programming, Dimitris Vyzovitis, MIT Media Lab

• GOO
  - www.jbot.org/goo