Metaprogramming

CS315B
Lecture 9
Projects

• Time to start thinking about projects!
  • Project proposal assignment is out today
  • A Regent or cuNumeric program/library of your choosing

• Working in teams is OK
  • But then it should be a more ambitious project!
What is Metaprogramming?

• Programs that generate programs

• Example: C++ template metaprogramming

• But a very old idea
  • Lisp in the 1950’s
  • Explored extensively since the 1980’s
Why Metaprogramming?

• Reason #1: Performance

• Consider a function $F(X,Y)$
  • $X$ changes with every call
  • $Y$ is one of a small set of possible values
  • Or fixed for long periods of time

• Generate versions $F_Y(X)$ for each value of $Y$
  • And optimize each $F_Y(.)$ separately
Why Metaprogramming?

• Reason #2: Software maintenance

• Maintaining versions $F_Y(X)$ for each value of $Y$ by hand is painful

• Much easier to maintain a program that auto-generates the needed versions
Why Metaprogramming?

• Reason #3: Autotuning
  • Based on performance measurements, generate a new version of $F(X)$
  • Here, machine characteristics are a “hidden”, constant parameter

• May need to generate many versions $F(X)$
  • Which versions and how many are data dependent
  • The space of possible versions could be very large or even infinite
Templates using Metaprogramming

• Templates are an instance of metaprogramming
  • Each template argument produces a distinct set of methods, customized to a particular type

• Lua can be used to generate Terra structs and methods
  • Example 32
Why Does this Work?

• Lua and Terra (and Regent) share a lexical environment
  • Lua variables can be referred to in Terra & Regent

• Terra types are Lua values
  • E.g., `Array(float)`

• In this example, can only have one `ArrayType`
  • The name can’t be redefined
  • Can also generate new names (not shown)
Escape

• Lua can also be used to compute Terra *code*
  • Expressions or statements

• The *escape* operator \([ e ]\) inserts the value of the Lua expression \(e\) into a Terra context
  • \(e\) is Lua code
  • That evaluates to a Terra expression

• Example 33 & 34
Warning! Warning!

• Metaprogramming is tricky

• It is easy to
  • Not get the code you expect
  • Perform illegal operations
    • E.g., adding two pieces of code, instead of two numbers

• Separate
  • Function definition time
  • Function call time

• Metaprogramming takes place at definition time
Guideline 1

• An escape operation [...] should contain
  • A call to a Lua function
  • An explicit quote `...
  • Not strictly necessary, but these are the common cases
Guideline 2

• To do metaprogramming, you will need both values and code at function-definition time
  • The values may appear in the final code
  • Or be used for computing the code

• Values that you use in metaprogramming
  • Must be defined at the Lua level
  • Outside of any Terra functions or Regent tasks
  • Examples 35-38
Metaprogramming in Regent

• Regent metaprogramming is similar to Terra

• Escape is still \[ ... \]

• Quote is `rexpr ... end`

• Example 39
  • New feature: A Lua function that returns a Regent task
Stencil_fast.rg

- A sophisticated example of Regent metaprogramming
Semantics

• It is worth understanding in some detail the semantics of metaprogramming in Lua/Terra/Regent.

• There are a number of steps ...
Semantics

• Step 1: Lua code evaluates normally until it reaches
  • a Terra/Regent function definition
  • A quote expression
Semantics

• Step 1: Lua code evaluates normally until it reaches a Terra/Regent definition or a quote

• Step 2: A Terra/Regent expression is specialized in the local environment, by evaluating all escaped Lua expressions
Semantics

• Step 1: Lua code evaluates normally until it reaches a Terra/Regent definition or a quote

• Step 2: A (Terra/Regent) quote is simply returned as code
  • Internally, a code data type
Semantics

• Step 1: Lua code evaluates normally until it reaches a Terra/Regent definition or a quote

• Step 2: The Terra/Regent expression is specialized in the local environment, by evaluating all escaped Lua expressions

• Step 3: When a Terra/Regent function is called, it is JIT compiled and returns a Terra/Regent code value.
Back To Step 2

• Step 2: The Terra/Regent expression is specialized in the local environment, by evaluating all escaped Lua expressions

• In this step, Lua/Terra/Regent share the same lexical environment
  • Escaped Lua expressions are evaluated
  • Lua variable references are replaced by their values
    • Must be coercable to a Terra/Regent value!
• Step 3: When a Terra/Regent function is called, it is JIT compiled and returns a Terra/Regent code value.

• Terra/Regent execute in a separate environment
  • All variable references are to Terra/Regent values
  • Can still call Lua functions, though!
    • Be careful
    • Will call into the local Lua interpreter on the node
Critique of Metaprogramming

• Most metaprogramming systems are designed to use language X to program in language X
  • Lisp
  • Scheme
  • MetaOCaml

• Plus
  • Expressive languages, easy to manipulate code programmatically

• Minus
  • Limits the performance that can be obtained
  • Because the languages are (usually) untyped, high-level, garbage-collected
Other Approaches

• Other approaches involve metaprogramming in lower-level languages through a variety of mechanisms
  • Template metaprogramming (C++)
  • Preprocessors (C)
  • Printf and recompile (C)

• Plus
  • Code can be as fast as possible

• Minus
  • Bizarre restrictions, cumbersome to use, not completely general
Metaprogramming with Lua/Terra/Regent

• Use a high-level language to metaprogram lower-level languages

• Plus
  • Generality, expressivity & performance
  • Key is shared lexical scope

• Minus
  • Need to understand two/three languages
  • Need to understand evaluation semantics
Lua/Terra for ATLAS

• ATLAS provides autotuned matrix multiply routines
  • Combination of X86 asm, C, C-preprocessor, Makefiles, custom scripts

• Terra version
  • *Staged* (metaprogrammed) Terra code
  • Autotuning written in Lua
    • Selecting optimal subproblem sizes for a machine
  • Optimizations: vectorization `vector(float,4)`, register blocking, cache blocking, unrolling
  • Total code is ~250 lines
ATLAS Results

![Graph showing performance comparison between ATLAS, Terra, Blocked Naïve, and Naïve for different matrix sizes in double precision. The graph indicates that ATLAS and Terra perform better than the Blocked Naïve and Naïve methods, with ATLAS maintaining a higher performance throughout different matrix sizes.]
Metaprogramming/Autotuning Regent

• Tune size/number of regions

• Tune depth of region tree
  • How many levels of decomposition is best?

• Specialize code to individual subregions
  • E.g., boundary vs. interior
  • E.g., repetitive sparse patterns

• Perform optimizations
  • But note the Regent compiler does some optimizations already
Summary

• Metaprogramming is a very powerful tool
  • You can program your own compiler functionality

• Not as exploited as it should be
  • And Lua/Terra/Regent makes it easier to use

• Give it a try!