CSE583: Programming Languages

David Notkin
7 March 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Tonight

- Domain specific languages
- A quick overview of programming language topics that we didn't cover
  - Any why
- Overview of final exam structure
- Course reviews

Domain specific languages

- These are programming languages focused on solving problems in a limited domain
  - By limiting the domain, the intent is to gain added leverage
- Overall, the goal of a DSL is to gain expressiveness at the cost of generality

Tonight

DSL = Domain Specific Language

Example DSLs

- CLP(R)
- Spreadsheets
- JavaScript
- csh
- make
- …more…?

- What domain does each focus on?
- What leverage does it get in that domain?
- What does it give up to gain that leverage?

Synonyms for DSL

- Little languages
- Micro languages
- Application languages
- Very high level languages
- Sometimes they are considered to be executable specification languages
  - Often highly declarative

Example: Unix shells

- Domain is
  - streams (e.g., standard in and out)
  - operations on streams (e.g., redirection and pipe into another stream)
  - processes (management)
- Simple control-flow and data (primarily string) manipulation mechanisms
- They are Turing-complete, but you'd never want to (for instance) manipulate complex data structures in the shell

Example: make

- Dependences between parts of a larger program are described declaratively
  - Technically, it needn't be a program, of course
- Actions to take when a simple temporal relation holds between two dependent parts are described imperatively
  - The description of these actions is outside the scope of make itself
- Domain details like file modification time and file suffixes are handled easily

Benefits of DSLs [IRISA/INRIA]

- Easier programming
  - Productivity gains of at least a factor of 10 have been cited
- Systematic reuse
- Easier verification
  - Higher reliability

Parameterization mechanism

- One can view complex parameters as their own DSL
  - Consider viewing the "little language" used as the string format for `printf` as a DSL
  - Data represented as a parameter ends up being a program to be processed
- This is a traditional tradeoff between program and data

Program families

- A program family is a set of programs that share enough in common that it is worthwhile to study them as a whole [Parnas]
- A program family provides an opportunity for developing a DSL from which it is easy (easier) to build instances of that family

Guideline

- If the similarities give you significant leverage, then you might consider a DSL to handle a program family
- Note: there are other software design approaches, such as layering, for handling program families, too

Guideline

- This guideline isn't so useful
  - The cost of developing the DSL isn't explicit
  - Presumably one intends to amortize this cost over both planned and future (unplanned) instances
- Aside: almost all decisions we make in software are roughly this ill-formed

How to design and implement a DSL?

- Pertinent to the cost issue, of course
- Who should develop DSLs
- Programming language design is difficult
- Programming language implementation is difficult
- The "DSL-ness" of DSLs doesn't make these less
  - Indeed, if you look at many DSLs, it's apparent it makes it worse!

Small example: reflexion mapping

- Last week:software reflexion models
- A key is the mapping from source code to high-level model
- We defined a little language for this
  - Used logical and physical structure in the source code
  - Used regular expressions

```
[file=scanner.* mapTo=Parse]
[file=buf\.[ch] mapTo=Parse]
[class=Parser mapTo=Parse]
[class=Ast mapTo=AST]
[class=AssignStmt mapTo=AST]
[class=BinOp mapTo=AST]
[class=Block mapTo=AST]
[directory=401 class=CallStmt
                    mapTo=AST]
```

Actually, a family of DSLs; each defines the structures for the given source code system

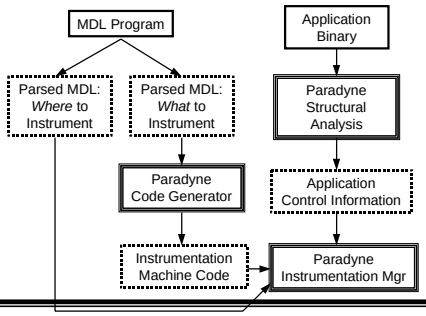
An obvious point

- Understanding language issues (and ideally, language design) will lead to better, more useful DSLs
- But be a bit careful about this point
  - As an example, TKL/TK is generally considered to be a lousy language
  - But it's clearly been useful and is used broadly

Larger example: MDL

- Paradyn is a project on tools for parallel performance at the U. of Wisconsin [Miller et al.]
- MDL is a "metrics description language", a DSL for instrumenting code

MDL control flow



MDL example

```
list pvm_msg_func is procedure {
   flavor pvm;
   items { "pvm_send", "pvm_recv" };
}
constraint procedure /Code is counter {
   append preInsn $constraint[0].entry
      (* procedure = 1; *)
   prepend preInsn $constraint[0].return
      (* procedure = 0; *)
}
```

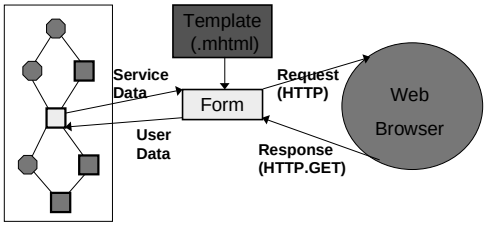

More of the example

```
metric msgs {
   name "Messages";
   units opsPerSecond;
   aggregateOperator sum;
   flavor { pvm };
// the base computation of the metric.
   base is counter {
   foreach func in pvm_msg_func
      append preInsn func.entry constrained
         (* msgs++; *)
   }
}
```


MAWL: another example [Atkins et al.]

"A form-based service is one in which the flow of data between service and user is described by a sequence of query/response interactions, or forms. Mawl is a domain-specific language for programming form-based services in a device-independent manner. We focus on Mawl's form abstraction, which is the means for separating service logic from user interface description, and show how this simple abstraction addresses seven issues in service creation, analysis, and maintenance."

MAWL structure



Greet.mawl

```
global int access_cnt = 0;
session Greet {
  local form {} -> { string id } GetName;
  local form { string id, int cnt,
               int time } -> {} ShowInfo;
  local int time_now = minutes();
  local string i = GetName.put({}).id;
  ShowInfo.put({i, ++access_cnt,
                minutes()-time_now});
}
```


GetName & ShowInfo.mhtml

```
<HTML><HEAD><TITLE>Get-Name Form</TITLE></HEAD>
<BODY>Enter your name: <INPUT NAME=id>
  </BODY></HTML>

<HTML><HEAD><TITLE>
   Show-Info Form</TITLE></HEAD>
<BODY>Hello <MVAR NAME=id>,
  you are visitor number <MVAR NAME=cnt>.<BR>
  Time elapsed since first form is <MVAR NAME=time>
  minutes.
</BODY></HTML>
```

Service creation, analysis & maintenance

- MAWL is not intended only to make writing form-based programs easier
- It's also designed to ensure that the engineering of these programs has specific benefits
  - compile time, implementation flexibility, rapid prototyping, testing & validation, support for multiple devices, composition of services, usage analysis

Compile-time guarantees

- A service should only generate valid HTML
- The HTML should be consistent with the service logic
  - That is, is the service prepared to handle the values entered by the user?
- Separating a service into sessions, forms, and templates enables such checking
  - The descriptions can be checked against one another
  - Most other approaches to generating HTML can't do this

Implementation flexibility

- A MAWL service can be compiled into a CGI script
  - This is easy, but may not scale to heavy hit rates
- It can also be compiled directly to an HTTP server
  - More complex, but may scale better
- MAWL handles this without modifying the service descriptions

Prototyping services

- Early versions of MAWL were statically type checked, requiring each MHTML template to always be typed properly
  - Programmers complained about this, since it compromised initial prototyping
- MAWL was modified to allow sessions to be compiled without templates at all (using a default)
  - This allows programmers to try out their service through forms more immediately

Testing and validation

- The separation of the pieces allows testing of the parts other than through a GUI
  - For example, a testing harness can provide an alternative implementation for each form's put method
- By separating control flow and state management from UI, exercising and analyzing the components is much easier

Multi-device services

- Separation into the separate components allows for making single services that can handle diverse devices
  - Standard browers, cell phone browsers, etc.
- Provide two different templates that have the same effect (e.g., mhtml and mpml, for phones)
  - The service sees a single consistent view
  - The differences are isolated in the templates

Composing web services

- Approach allows linking to existing web pages
- Can also combine information from other pages
  - i.e., like the Metacrawler
- Data from these other pages can be treated like MAWL user data

Usage analysis

- MAWL forms provide a centralized point for monitoring interactions between the service and its users
  - A form put method is instrumented to record service data
  - (A visualization tool is provided, too)

MAWL

- Collectively, the MAWL language design shows the leverage one can get from a carefully designed DSL
  - Types
  - Separation of control and data flow
  - Separation of concerns
  - Other language design issues?

Key issues: given a DSL design

- Should you implement it as a standard language is implemented?
- Should you implement it in the context of an existing language?
  - Haskell, monads, etc.
- How do you get all the associated tools (debuggers, monitors, visualization, etc.)?

No simple answers

- The question is (as are many software engineering questions) largely an economic one
  - What is the cost of developing the tools in different ways?
  - How does it affect the quality of the tools?
  - How does it affect time-to-market?

Bottom line on DSLs

- They are everywhere, even when you don't think of them as DSLs
- Thinking about them as languages gives a ton of leverage
- The implementation issues are equally complex as the design issues
  - And beyond the scope of this course

Topics we didn't cover

- And why
  - But you'll have to listen to the lecture

Programming language semantics

- Everything beyond the BNF
- Operational semantics
  - Abstract interpretation (fancy data flow)
- Axiomatic semantics
  - Pre/post conditions to define constructs
- Denotational semantics
  - Associate abstract syntax with semantic domains

Partial evaluation, program specialization, run-time compilation

- Building faster programs through delayed binding
- Ray tracing efficiently written in C ran 1.5 to 3 times faster after specialization to the scene to be drawn
- A Fortran FFT program was specialized with respect to a fixed function and number of points, running x1.5-4
- Compiler generation from interpreters for programming languages

Programming calculi (Other than the λ-calculus)

- Reasoning about programs
  - CSP
  - CCS
  - Π-calculus
  - Several for handling mobility (for the web)

Parallel and concurrent languages

Logical frameworks and automatic deduction

Functional logic programming

Literate programming

What else?

Final

- Functional languages 25%
- Object oriented languages 25%
- Logic and constraint logic programming 25%
- Miscellaneous 25%
  - Language design issues, visual programming/program visualization, domain specific languages, etc.

Final

- No code to write
  - Well, no code to write and execute
    - It's possible I'll ask for snippets of code
- Concise and clear answers
  - Increased partial credit

Final

- Available on web tomorrow, Wednesday March 8, by 5PM
  - I'll send email when it's ready
  - It'll be in PDF form
- You can choose any consecutive 3-hour period to do the exam
- It is due by 11:59PM PST on Sunday March 12, 2000
- No electronic turn-in!
  - Get it to me (by snail mail, under my door, in my office mail box) in time