Extensible Languages

Reflection and Meta-Programming

Motivation

Why care about extensible languages?

We can express many domain-specific languages (or policies) as language extensions

Many benefits

• Technical
  - no need to re-implement language constructs (if, while, functions, records, etc.)
  - extensions only need to be transformed to existing constructs
  - decreased development costs

• Economic
  - environment, tools (editor, debugger, documentation tools) can be reused
  - decreased transition (project adaptation) and education costs

Motivation: Domain-Specific Languages (DSLs)

DSLs result in significant productivity increase

• domain knowledge captured in language

• reusable, general, efficient form

Boundaries of languages-libraries not exact

• practically, every reusable library that is more than a collection of functions can be viewed as a new domain-specific embedded language (e.g., STL, MFC)

• is an OO framework a language or a library?

• no strict separation => no strict comparison

Many technical advantages of a well-designed DSL over a library of functions

• Simpler, intuitive syntax

• Higher level primitives

• Possibility for higher-level optimizations
  - e.g., query optimization in database languages

• Advanced error-checking
  - error checking of functions is only type checking of operands

A tremendous number of libraries for special purposes

• >1900 special-purpose APIs from Microsoft
Extensible Languages Classification

Language extensions can be

- **Syntactic**: new syntax is added to the language (e.g., macros)
- **Semantic**: no new syntax is added but the semantics is changed (e.g., meta-object protocols)

Two main approaches to language extensibility (not strictly divided):

- **Transformational**: the meaning ("semantics") of an extension determined by syntactic transformations to more basic language primitives
- **Compositional**: the meaning of an extension is determined by directly manipulating (appropriately externalized) internal structures of the compiler

Usually (but not always) we associate the terms

- *meta-programming* and *transformation system* with transformational extensibility
- *reflection* with compositional extensibility

More specifically

**Meta-programming**: the act of writing programs that (re-)write other programs (e.g., macros)

**Reflection** (in the context of languages): the act of a language allowing access to its internal functionality

Also,

- the "meta" prefix commonly used for most reflective activities (e.g., meta-object protocol)

Semantic extensibility is really ill-defined

- when is something a semantic extension and when a regular program?
- when is a language construct "reflective"?
- grey areas (e.g., first class continuations, OO messages, etc.) but usually we can draw a line intuitively

We will review several language extensibility mechanisms (there are many more but these should illustrate the ideas)

- CLOS, SOM, Java Reflection, Intentional Programming, Open C++, JTS, Lisp and Scheme macros

Semantic Extensibility

- No new syntax. Semantics (policy) changed
- Best known examples: meta-object protocols

Meta-object protocols (MOPs):

- associate semantic changes to a class with a class meta-object (run-time MOPs)

The meta-object’s class (meta-class) has methods defining extensions for various semantic actions

- the choice is arbitrary (why not a set of meta-functions?) but shows good object-oriented design structure
Example: CLOS MOP

- CLOS is an object system for Lisp
- Provides semantic extensibility (both transformational and compositional) through a very powerful MOP
- Transformational character provided by the Lisp meta-programming facilities
  - code expressions as lists, quote, backquote, and comma

CLOS MOP compositional capabilities:
- can define before-, after-, and around-methods
- can change (multiple) inheritance policies
  - how to inherit, what to inherit, how to mix members, inheritance precedence, how to combine methods (e.g., superclass method runs first like in Beta), etc.

Simple example:

```
(defclass counted-class
  (standard-class)
  ((counter : initform 0)))
```

counted-class is a meta-class (its superclass is the standard meta-class standard-class).

Every object of counted-class (in essence, every class created with counted-class as its meta-class) will have a counter field

```
(defclass foo () () (:metaclass counted-class))
```

Class foo is associated with a class meta-object whose class is counted-class. This is equivalent to saying “foo’s meta-class is counted-class”

[example continued]

```
(defmethod make-instance :after
  ((class counted-class) &key)
  (incf (slot-value class 'counter)))
```

make-instance is the method of a class that creates a new object

Here we create an after-method for instances of counted-class

Recall that class foo is (or more correctly “is associated with”) such an instance

Hence, every time a new foo object is created, foo’s counter is increased by 1

CLOS MOP transformational capabilities:
- strictly speaking, CLOS does not deal with code transformation
- but its reflective capabilities work nicely with Lisp program manipulation

Example:

```
(defun generate-defclass (class)
  `(defclass ,class-name class
    ,(mapcar #'class-name
      (class-direct-superclasses class))))
```

Gets the names of all superclasses of a class and generates a class definition (in source code form) for a class with these superclasses
Example: SOM (IBM's System Object Model)

- SOM is a binary object system and offers a meta-object protocol for industrial languages (C, C++, ...)
  - something like COM
  - this ensures binary compatibility under object evolution—even for MOP issues

- Semantic compositional approach

- Model similar to CLOS (classes are instances of meta-classes)

- Classes specified in SOM IDL (interface definition language—CORBA compliant)
  -C, C++ header files produced and executed programs use the SOM runtime
  -dynamic class construction
  -extra level of indirection allows binary compatibility

• SOM has nothing to do with the C++ object system
• SOM meta-classes are mapped to C++ classes when C++ is the host language
• SOM classes are dynamic entities (objects)

```java
interface Counted : SOMMCooperative {
    readonly attribute long counter;
    implementation {
        somMethodProc** doNew;
        somInit: override;
    }
};
```

This is the interface definition of the meta-class and its (SOM-specific) implementation

Regular class definitions are simple IDL definitions with a metaclass field assignment in the implementation section (see above)

Interesting issues specific to SOM (example):

*Metaclass Incompatibility*

- A class X has a metaclass XMeta, depends on a method of its metaclass (methods of a class can call methods of their metaclass)

- A class Y inherits from X, but specifies a metaclass explicitly (YMeta); problem

- Solution: SOM automatically builds a metaclass DerivedMeta for Y, which multiply inherits from XMeta and YMeta
  - what if methods conflict in XMeta/YMeta?
  - Usual multiple inheritance caveats apply. A “solution” in OOPSLA’94 paper (“Reflections on Metaclass Programming in SOM”)

- This technique is the cornerstone of binary compatibility: the user does not need to worry about metaclasses when the library changes (e.g., the metaclass of a library class changes)

Example: Java Reflection Classes

- No extensibility—merely introspection
- class meta-objects like in CLOS, SOM (instances of Java.lang.Class)

- allows dynamic inspection of the class of an object and its inheritance hierarchy
- allows dynamic loading and linking of classes

- mainly geared towards object inspectors, debuggers, class browsers, interpreters, etc.

- could become quite interesting with a few extensions:
  - allow manipulation of the inheritance hierarchy?
  - give access to method bodies, even in opaque form?
Syntactic Extensibility

- Syntactically extensible languages allow the specification of new syntax
- Pure compositional extensibility is limited in certain well-defined aspects of a language - the implementors of the language must anticipate all extensions
- This is why most syntactic extensibility mechanisms have a transformational part
- Transformational extensibility works by transforming extensions to basic language primitives - Obviously, macro expansion is a special case
- In theory, transformational extensibility is very powerful

In practice, some extensions are very hard to express as transformations alone - some “semantic” information needed (types, blocks, etc.)

Often the two kinds (transformational and compositional) of extensibility are combined for more power

More on this later...

Example: Open C++

- A transformational (compile-time) MOP!
- Limited syntactic extensibility, more powerful semantic extensibility
- Only new syntax that can be added:
  - type modifiers (like “static”)  
  - access specifiers (like “private”)  
  - “while” and “for”-like statements  
  - “function” like blocks of code
- Code representation like in Lisp: parse trees represented as nested linked lists
- Can create new trees, pattern match on trees, etc. (standard set of operations)
- Simple introspection protocol on trees representing classes (can examine members, fields, superclasses, metaclasses, etc.)

Semantic extensions can be specified for translating classes, members, methods, method calls, and many more

Simple example:

```cpp
metaclass Person : MyMetaClass;
class Person {
    int age;
    public:
        Person(int age);
        int Age() {return age;}
};
```

Specify that MyMetaClass is the meta-class for class Person
Example: JTS (Jakarta Tool Suite)

- Syntactic transformational extensibility mechanism
- Main element: syntactic extensions specified as new productions in context free grammar
- Extended grammar defined as the union of original productions and extension productions
- Extensions are layered — new languages formed by selecting extensions organizing them in a type equation
- Meta-programming model: abstract syntax trees, code templates, pattern matching, hygienic constructs (more on that later)

Example: Lisp and Scheme Macros

- Syntactic transformational approach
- Languages of the Lisp family have simple syntax
- Easy to manipulate source code programmatically, extend syntax
- The term “macros” does not necessarily refer to pattern-based macros (as in C)
- Lisp has programmatic macros (general meta-programming)
- Scheme has two (proposed) macro mechanisms:
  - high level (hygienic, pattern-based)
  - low level (programmatic, compatible with high level, many proposed)

Programmatic macros example (Lisp)

(defmacro send-passwd (string)
  '(send-to-host
    (decrypt , (encrypt string))))

Usage:

(send-passwd "gandalf13")

Converted after macro-expansion into:

(send-to-host (decrypt "09871230123481234")

- That is, the password never appears decrypted in the object file.
- Gets encrypted at compile time (rather, macro-expansion time), decrypted at run-time!
- Can’t do this in C
- Note: Lisp makes no distinction between code and code as data when it comes to constants
  '1 = 1