A Practical Example of Language Design and Implementation

Anya Helene Bagge

Bergen Language Design Laboratory, University of Bergen

Software Language Engineering Course
Koblenz, 2013-01-08
Magnolia Overview

The language for flexible, reliable software!

Programming with...

- Abstraction
- Specification
- Separation of Concerns

We’re doing:

- Software analysis & transformation
- Model-driven – except we don’t use the terminology

Main design focus:

- Ease of reasoning!
  - ...for you, for me, for the compiler
  - Specification is integrated with code
- No aliasing
- No higher order
- Highly structured iteration
- No OO/dynamic dispatch
- Control over resources

But also flexibility, reliability, maintainability, extreme code reuse.
Magnolia Programming

- Supported by an Eclipse plugin
- Current backend target language is C++
- All the basic stuff in the language are libraries
  - Implemented in C++
- Typical story
  - Design your interfaces, building on existing ones
  - Specify behaviour with axioms
  - Provide or reuse implementation(s)
  - Test using automated axiom-based tests
- Reuse support
  - Everything can be renamed, declarations added/dropped, parameters changed, etc.
- Meta-programming
  - To be added
  - Parts of compiler API is exposed and can be called at compile time
Concept-Based Programming

Integration of specifications in programs:

- Ideas are modelled as concepts
- Each concept is a set of abstract types, operations, requirements and axioms – corresponding to an algebraic specification
- Concepts can build on (refine) other concepts
- A concept may have several interchangeable implementations

Motivation

- Making semantics available to the compiler (axioms)
- Separating specification from implementation
Conceptual Overview
Example Concept: Dictionary

concept Dictionary(Dict, Key, Val) {
    type Dict; type Key; type Val;
    requires EqualityComparable[T => Key];

    function Dict create();
    function Dict put(Dict, Key, Val);
    function Val get(Dict, Key);
    function bool contains(Dict, Key);
    function bool isEmpty(Dict);

    axiom dict1(Dict d, Key k, Val v) {
        assert get(put(d, k, v), k) <-> v;
        assert contains(put(d, k, v), k) <-> true;
    }
    axiom dict2(Dict d, Key k, Key l, Val v, Val w) {
        if(k != l)
            assert get(d, k) <-> get(put(d, l, w), k);
    }
}
Dictionary – Different Implementations

**Dictionary**
- `create()`
- `put(d, k, v)`
- `get(d, k)`

**HashMap**
- `= create()`
- `= put(d, k, v)`
- `= get(d, k)`

**Search Tree**
- `= new()`
- `= insert(k, v, d)`
- `= lookup(k, d)`

**List-of-Tuples**
- `= List((type K, type V))`
- `= List((k, v), d)`
- `= if head(d)[0] == k` then `head(d)[1]` else `get(tail(d), k)`
Every imperative procedure

```plaintext
procedure _++_(upd a : string, b : string);
```

has one or more corresponding algebraic functions:

```plaintext
function _++_(a : string, b : string) : string;
```

Only the imperative version needs to be implemented – uses of the functions are automatically mutified to imperative code

```plaintext
var s = "Hello, " ++ person ++ "!";

var s = "Hello, ";    // s = "Hello, 
call _++_(s, person);  // s = "Hello, " ++ person
call _++_(s, "!");    // s = "Hello, " ++ person ++ "!
```
Mutification & Functionalisation

Motivated by

• Intuitiveness of algebraic style
• Direct relationship with axioms / algebraic specifications
• Performance benefits of imperative style vs algebraic style

Evaluation

• Positive experience with Sophus, implemented in algebraic style:
  • 1.8–2 times speedup of algebraic code (6–8× with rewrite rules)
• Enables...
  • easy application of rewrite rules and optimisations
  • easy integration of specification and implementation
  • simple data-flow analysis and dependency analysis
  • convenient way of dealing with multi-result operations by specialisation
A Small Example

In Java:

foo.sort()

What happens to foo?

foo.sort(cmp)

Will cmp be changed?

In Magnolia:

foo = sort(foo);

We know what happens to every parameter.

foo = sort(foo);
bar = search(foo, name);

We know that foo is sorted, and can choose binary search

foo = sort(sort(foo));

We know that the outer sort is useless
Example: Pairs of Numbers

PairExtensions.mg
Implementation
Language Services

- Build/compilation
- Error/warning squiggles
- Hover Help
- Navigation
- Outline
- Project Overview
- Refactoring
- ...

Accessing the IDE Infrastructure

Main entry point:

- **Workspace manager**

The Workspace manager gives you a **Project manager**

- Maintains mapping between files/URIs and Magnolia packages
- Maintains a dependency tree
- Listens to changes in the workspace
- Is thread-safe

Available in both Eclipse and non-Eclipse versions

- Could run in a separate process, and provide services to any IDE
- Works with Emacs
Compiler Organisation

• Frontend: parsing, desugaring, “loading”
• Typechecking: flattening concept code, typechecking normal code
• High-Level Optimisation: not implemented yet
• Backend: for each library unit
  • Mutify, remove any other “special” Magnolia constructs
  • Restructure to C++-like code
  • Pretty-print to C++; add standard stuff
Compiler Examples

- Desugaring
- Flattening
- Expression typechecking
- C++ translation
Facts
Facts

A fact is a piece of non-trivial information:
- Computed and stored independently
- A handle to a (large) piece of data

Each fact has a signature:
- Computed from dependencies

Facts may be in one of three states:
- Available, and up to date (based on signature)
- Available, but out of date
- Not available
Fact Data and Storage

Fact data may be:
- Stored using soft/weak references
- Automatically stored to disk

Each fact may be connected to a **store object**
- If a fact is unavailable, it will attempt to read from the store
- If a fact is updated, it will send data to the store
**Fine-Grained Dependencies**

Facts are fairly coarse grained, e.g.:

- The list of modules in a package
- The typechecked AST of a module
- The environment/symbol table at the top-level of a file

For fine-grained stuff, we use **memoisation**:

- \( f = \operatorname{memo}(f\text{Impl}) \) makes a memoised version of the Rascal function \( f\text{Impl} \)
- Results of calls to \( f \) is stored in a hash table indexed by the arguments
- Compiler library uses pure functions for easy memoisation
- Speedup: 2x for desugaring, 1.05x for simple operations on names
- Memoisation can be basis for multithreading
The IDE
IDE Interaction Model

- Resource management system
- Compiler library
- Facts
- Small Eclipse-specific layer

Example: Hover Help
- Find package
- Ask for xrefs
- Present info

Q: How to map location to info?
Language Services (again)

- Build/compilation
- Error/warning squiggles
- Hover Help
- Navigation
- Outline
- Project Overview
- Refactoring
- ...

...
Refactoring: Inlining

Basic recipe:

- Find something you want to inline (e.g., by location)
- Apply the inlining
- Pretty-print the inlined code, patch it back into the surrounding source text
More Fun with Inlining

- Rename constant, variable, type...
- Change method declaration
- Introduce wrapper
More Stuff
Axiom-Based Rewriting

Each equational axiom is a potential rewrite rule:

- Choose one side for matching, and the other as a replacement

**Examples**

unwrap(wrap(x)) <-> x
x * (y + z) <-> x * y + x * z
if(sorted(A))
  sort(A) <-> A

For axioms to be useful in rewriting, we must know

- **Which** axioms are useful
- **When** they are useful
- **What** they are useful for

Such strategy information can be provided by axiom classes:

- Simplification, propagation, traversal order, etc
Other Thoughts

• Energy Efficiency
• How multi-threaded can we be?
  • Does that even matter if we hand-code in Java?
• Performance
Conclusion

It’s all:
- Implemented in Rascal (language stuff) and Java (IDE stuff)
- Organised as a library

Magnolia IDE offers:
- Hierarchical project overview
- Soft-referenced storage of facts / intermediate result
- Library of compiler operations
- Automatic disk storage
- Fairly smooth experience on 8 cores, but still slow...