Unifying execution of imperative and declarative code

Martin Kožený
Faculty of Informatics
Czech Technical University in Prague
Czech Repuplic
Email: martin.kozeny@gmail.com

***Abstract*—Using unified environment for both imperative and declarative code is quite interesting point of view on programming these days. Why is it interesting? Obviously by ability of mixing imperative and declarative code can programmer easily express constraints of problem using declarative code in terms of existing data structures in imperative programming. This give us very strong tool to solve easily problems, which are especially in imperative programming difficult to solve.**

I. Introduction

Motivation for writing this paper is to present unconventional way of programming, show how it works and its advantages and disadvantages and sketch its possible development in the future.

First question when solving problem of unified execution environment is how to mix declarative and programming paradigms together. What are the possibilities of interleaving or cooperating between these two paradigms. Regarding this issue, there are two different attitudes.

First one is Java framework Squander developed at Massachusetts Institute of Technology by Mr. Aleksandar Milicevic. This framework implements declarative paradigm in prgramming language Java using annotations. Via those annotations can programmer declare preconditions of heap state, part of the heap, which is allowed to modify and how should objects on heap look like after execution.

Another approach provides Jess rule engine. This engine was entirely written by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, CA. Jess uses enhanced version of the Rete algorithm to process rules. Rete is a very efficient mechanism for solving the difficult many-to-many matching problem. Jess engine consist of working memory representing set of facts and rules for manipulating with those facts. Engine can directly access Java objects, invoke methods or implement interfaces.

II. Framework Squander

Squander is a framework that provides a unified environment for writing declarative constraints and imperative statements in the context of a single program. This is particularly useful for implementing programs that involve computations that are relatively easy to specify but hard to solve algorithmically [1].

Let us define execution of Squander on framework's architecture presented at Figure 1.

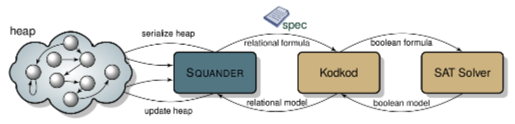

Fig. 1. Architecture diagram

Excution of Squander is defined by following steps [2]:

1) serialize heap into relations
2) translate specs and heap relations into Kodkod
3) translate relational into boolean logic
4) (if a solution is found) restore relations from boolean assignments (if a solution is found) restore field values from relations
5) (if a solution is found) restore the heap to reflect the solution

Main terms defining this framework are Kodkod and JSFL.

Kodkod is a solver for relational logic. Kodkod requires bounded universe, a et of untyped relations, bounds for every relation and relation formula. Then translates given problem into boolean satisfiability problem (SAT) and applies of-the-shelf SAT solver to search for satisfying solution, which is reflected back if found.

JSFL is formal lightweight specification for Java supporting relational and set algebra, as well as common Java operators. Using expressive power of relation algebra, JFSL makes it easy to succinctly and formally specify complex properties about Java programs such as method pre and postcondition, class invariants and so called frame conditions, which means portion of the heap, that is methods allow to modify. It also supports specification fields which can be useful for specifying abstract data types.

III. Jess

This section is written using Jess manual [4].

Jess is a rule engine and scripting environment using form of declarative rules and is a highly specialized form of Lisp. The payoff of this engine is that it is very expressive, and can implement complex logical relationships with very little code.

In Jess everything is a function call. Each Jess rule engine holds a collection of knowledge nuggets called facts. This collection is known as the working memory. Working memory is important because rules can only react to additions, deletions, and changes to working memory. You cannot write a Jess rule that will react to anything else.

In Jess, there are three kinds of facts: unordered facts, shadow facts and ordered facts. Unordered facts provide slots, where data appeared similarly like fields in Java objects. Shadow facts are connected to Java objects, so by using shadow facts is possible to put any Java object into Jess's working memory. In that way, template for the fact is created by looking at a Java class. When some property of the Java object respectively Jess fact is changed, it is necessary to explicitly notify Jess fact respectively Java object about this change. Ordered facts are simply Jess lists, where the first field (the head of the list) acts as a sort of category for the fact.

Rules are used to search the facts to find relationships between them, and also can take actions based on the contents of one or more facts. A Jess rule is something like an *if then statement* in a procedural language, but it is not used in a procedural way. While if... then statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their if parts (their left-hand-sides or LHSs) are satisfied, given only that the rule engine is running. This makes Jess rules less deterministic than a typical procedural program.

The obvious implementation for the rule engine is very inefficient. This implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the working memory and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the working memory is stable, most of the tests will be repeated. You might call this the rules finding facts approach and its computational complexity is of the order of $O(R.F^P)$, where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the working memory.

Architecture can be many orders of magnitude faster than an equivalent set of traditional if... then statements. This is caused by using RETE Match algorithm for pattern matching, where the inefficiency described above is alleviated by remembering past test results across iterations of the rule loop.Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like $O(R.F.P)$, or linear in the size of working memory.

Jess rules has two parts. The first part consists of the LHS pattern. The second part consists of the RHS action, which is executed, when LHS part of the rule is fullfilled. LHS of a rule consists of patterns which are used to match facts in the working memory, while the RHS contains function calls. The LHS of a rule (the "if" part) consists of patterns that match facts. The actions of a rule (the "then" clause) are made up of function calls.

IV. Conclusion

This paper describes different approaches of implementation of unified environment for declarative and imperative code. Framework Squander provides this environment using Java built in annotations whereas Jess uses special scripting environment. Main difference is that Jess rules engine was primary developed for building complicated expert systems [3] like diagnostic systems whereas Squander was invented for solving hard constraint problems or generate data structure instances satisfying complex constraints. Moreover Sqaunder was developed in recent years and nowadays is used for further academic research of this topic but Jess was written almost twenty years ago and many systems was developed using this engine.

References

[1] Aleksandar Milicevic; MASSACHUSETTS INSTITUTE OF TECHNOLOGY, Department of Electrical Engineering and Computer Science; *Executable Specifications for Java Programs*, September 2010.

[2] Squander official website;
`http://people.csail.mit.edu/aleks/squander/`

[3] Ernst Friedman-Hill; *Jess in Action*, September 2003.

[4] Jess official website;
`http://www.jessrules.com/jess/docs/71/`