Project Plan

Team Dec 15-12: @PaniniJ
April 25th, 2015

Advisor	Dr. Rajan	
Client	Dr. Rajan	
Team Members	Dalton Mills	*Webmaster*
	David Johnston	*Team Lead*
	Trey Erenberger	*Key Concept Holder*

Abstract: This project's main goal is to provide Java developers with a simple way to safely write concurrent programs using the Panini capsule-oriented programming paradigm. This is achieved by producing a deliverable `.jar` file that will easily integrate with standard Java workflow and toolchain. The annotation processor in this `.jar` will be triggered by any standard Java compiler to generate capsules and other artifacts needed for their operation. These capsules will run concurrently in a Panini capsule system.

Table of Contents

Project Statement

Modern multi-core CPUs provide powerful support for concurrent programs, but it is very hard to build correct programs and systems which take full advantage of this hardware. PaniniJ is a Java-like programming language with strong support for the capsule programming abstraction. The language has been designed such that programs written in the language cannot include certain common concurrency bugs.

Though `panc`, the existing PaniniJ compiler, is an extension of the standard Java compiler, PaniniJ code is not easily integrated into existing Java projects. Our project, @PaniniJ, intends to make Capsule-Oriented Programming more accessible to Java programmers than the existing PaniniJ tools can provide.

Our project will implement an alternative mechanism for generating capsules, one which can be used by a Java developer within the Java programming language and built using standard Java build systems (e.g. Ant, Maven, Eclipse and `javac`). Our solution will use annotation processors to easily generate capsule classes which are functionally equivalent to PaniniJ capsules but usable from within a Java development ecosystem.

Prior Work: Background on Capsule-Oriented Programming

Concurrent programs are different from sequential programs because they must take into account manual management of memory accesses by using locks or other synchronization mechanisms. This management leads to lots of standard boilerplate code that must be duplicated wherever memory synchronization is needed. This style of programming has a high barrier of entry. In traditional concurrent programs and programming languages, the synchronization logic and the application logic are intertwined in a way which complicates reasoning, readability, and maintainability.

One alternative to lock-based (e.g. Java, C++) concurrent programming is actor oriented programming. Capsule oriented programming is a programming model refines the actor oriented programming model by placing certain restrictions on the programming model in order to provide certain safety properties.

Capsules differ from actors in that they have a finite list of messages that can be passed to them. These messages must all be defined statically and can be checked and verified formally at compile time. The syntax for sending a message looks like the syntax for calling a method call in Java. Capsules systems are statically defined whereas actor systems are

dynamically defined. The rigidity provided to capsule systems allows for formal proving of systems.

Usage: Capsule Generation via Annotation Processor

Annotation processors provide a simple way to hook into a standards-compliant java compiler, inspect annotated Java code, and then automatically construct additional Java source files. These machine-generated Java source files are then compiled and included for use in other source code in the project.

Our solution will be distributed as a `.jar` file which users can easily include in their java project. The user can then write a Java class annotated with `@Capsule` and suffixed with `Template`. This original `@Capsule`-annotated class is referred to as a *template class* because it serves as the template from which a capsule class is generated. When this class is compiled, `javac` will automatically call on our annotation processor, which will then inspect the template class and generate a variety of Java source artifacts.

Arguably, the most important of these generated source artifacts are the capsule classes, of which there are four types (Thread, Task, Monitor, and Serial). Of these four types, the Thread-based capsule is the canonical example (and the focus for our implementation in the first term).

Additional information about the design of capsules and our system which is used to generate them can be found the the design document.

Requirements

The requirements listed here are expanded on in the Design Document.

Functional Requirements

- Working annotation-based frontend for the generation of PaniniJ-compatible capsules.
- Support for four basic execution strategies (i.e. Thread, Task, Monitor, and Serial).
- Port PaniniJ programs to @PaniniJ programs for use as
    - capsule-generation test cases
    - semantic analysis test cases

    - benchmarks comparing PaniniJ and @PaniniJ performance.
- Intra-capsule confinement analysis.
- Analysis of other safety and correctness properties.
- Automated selection of appropriate execution policy based on system topology and capsule performance.
- Reference-counting garbage collection.
- Works with standards-compliant Java compilers.
- Works with common Java toolchains.

Nonfunctional Requirements

- Capsule template syntax is simple and declarative.
    - Minimizes amount of boilerplate code
- Validation and Acceptance Test for each functional requirement.

Solutions Considered

We investigated three possible solutions that would allow us to make capsule programming more accessible:

Option 1. Eclipse Plugin / Modify Eclipse

Many language development teams create an integrated development environment program quickly by extending Eclipse to support custom language syntax and features. The process of extending eclipse is made easier through the use of XText[1]. XText is a library that facilitates creation of Eclipse plugins and language development simpler.

This option would require more maintenance in order to stay relative. Since we would be building on top of Eclipse, our forked editor or plugin would be locked with the version of eclipse we started with, and we would need to perform some merges when Eclipse is updated.

This option would require end-users to download the custom editor or plugin. This could be a potential deterrent from software developers using PaniniJ to create multithreaded programs.

Option 2. Modify Eclipse JDT for PaniniJ

[1] https://eclipse.org/Xtext/

This option was only briefly explored. However, it would be similar to option 1 such that maintenance would be required in order to keep up with the latest changes in Java.

Option 3. Annotation Processor

Java provides the ability to have custom annotations through creating an annotation processor(s). Java developers are already familiar with annotations such as `@override` and `@suppress`. We would a create a handful of new annotations along the lines of `@Capsule`. When compiled, any user code containing our custom annotations would get processed and we would generate code similar to the Java source artifacts produced by panc.

Solution Chosen

We opted for Option 3 of creating a service built on Java's annotation processing. The other options were discarded fairly quickly in favor of this approach for the reasons outlined below:

1. Annotation Processing is now standard in Java. We would not be creating a non-standard solution.
2. Creating a standard solution caters to our ultimate goal of having accessible capsule-oriented programming.
3. Having a standard solutions means more maintainability and compatibility with future versions of Java.
4. It will be easier to integrate into existing projects.
5. Java developers are already familiar with annotations.

Feasibility Assessment

Risk reduction was the ultimate deciding factor in why we chose to use annotation processing for this project. Moving forward, however, we still have many concerns regarding potential and inherent risks of our proposed solution, as well as questions about the feasibility of achieving our goal of accessible capsule-oriented programming. An important part of our project is answering these questions.

Performance

How will multiple stages of code generation and type checking affect compiler memory and speed performance?

We don't know. Ultimately using @PaniniJ will lead to slower compile times compared to standard Java due to the layers of generation and compilation. The optimization of compiler memory and speed has not be a priority for this project.

Will code generated by the annotation processor be as performant as `panc` generated code?

Runtime performance is essential to the success of @PaniniJ as the primary advantage of concurrent programming is increased speed. Great care has been taken to reduce the number of method calls in the generated code in order to keep the overhead low enough for @PaniniJ programs to compete with sequential programs.

Usability Questions

- Will @PaniniJ lower the barrier of entry for developers new to programming concurrent solutions?
- Will @PaniniJ provide sufficiently familiar development workflow for the average developer programming concurrent solutions?
- @PaniniJ template and capsule classes are just Java code, but what will be required from the user to make common Java IDEs work seamlessly with @PaniniJ?
- Will IDE debugger integration be possible? If so, will it work consistently across all IDEs for which @PaniniJ is implemented?
- Will @PaniniJ be able to provide the semantic analysis expected by experienced IDE users? If not, how much will this negatively impact the user's ability to program effectively using @PaniniJ, and thus their desire to use @PaniniJ at all?

Schedule

By the end of the Spring 2015 semester:

- Generation of Capsule and Signature artifacts from annotated template classes
- Generation of Duck Future artifacts to support generated capsules
- Support for Thread capsule execution strategy.
- Port some existing PaniniJ programs to @PaniniJ.

- Benchmark ported @PaniniJ programs and their performance to original PaniniJ programs.

By the end of the Fall 2015 semester:

- Implement unfinished features of the core @PaniniJ prototype, such as, Capsule arrays, array return values, and primitive return values.
- Port more PaniniJ benchmark programs to @PaniniJ for use as tests and benchmarks.
- Investigate means of implementing semantic analyses to test for additional safety/correctness properties.
- Support for Task, Serial, and Monitor capsule execution strategies.
- Automated selection of appropriate execution policy based on system topology and capsule performance.

Cost

This project does not have any hardware components and does not require the purchase of any commercial software. By building on top of research undergone at Iowa State University we receive a vast pool of theoretical and technical knowledge that we can draw on with no cost. We expect our project budget to be $0 since there is nothing to be manufactured and we are working for free.

Market Survey/Literature Review

Our final product will compete with existing PaniniJ tools created within the Laboratory for Software Design at Iowa State University, in particular, the `panc` compiler. Our products are comparable because they both provide support for capsule-oriented programming. However, the existing solution has several restrictions that negatively affect its usability for the average Java programmer. Our project seeks to resolve some of these shortcomings.

PaniniJ currently inhibits the programmer from using standard Java toolchains. A client needs to use the special `panc` compiler. However, our solution is easily integrated via the classpath by hooking into the standard Java compiler produced by Oracle. `panc`, on the other hand, is a fork and extension of the Java compiler produced by Oracle. Our product decouples the capsule-generation process from a specific version of compiler allowing for portability across toolchains, compilers, and compiler versions.

Both systems produce comparable capsule systems. The code which our system generates is directly modeled after the code generated by `panc`. We have taken advantage of several

optimizations that reduce the overhead cost of using @PaniniJ when compared to a sequential programming model by using the output of `panc` as a guide for the output of @PaniniJ.

Project's Future

After our work is completed, the project will be released as an open source project on Sourceforge or Github. It is expected that it will then be maintained and contributed to by the Iowa State University Laboratory of Software Design's Panini research group.

If our project is successful and we implement enough functionality to make our product functionally competitive with the existing `panc` compiler implementation, it is expected that @PaniniJ will replace PaniniJ and `panc` in future Panini capsule oriented programming research. This would mean that the Panini research group would build on @PaniniJ in order to add additional features to @PaniniJ as the Panini programming model expands.

Glossary

Panini	A capsule-oriented programming language whose goals are to ease development of correct, scalable, and portable concurrent software.[2]
PaniniJ	A capsule-oriented extension of the Java programming language that runs on the standard JVM platform.[3]
Capsule	Similar to a process, it defines a set of public operations and also serves as a memory region for some set of ordinary objects.[4]
Capsule-oriented Programming	A programming paradigm that aims to ease development of concurrent software systems by allowing abstraction from concurrency-related concerns; entails breaking down program logic into distinct parts called *capsule* declarations and composing these parts to form the complete program using *system* declaration.[5]
Java annotations	A form of metadata, provide data about a program that is not part of the program itself, having no direct effect on the operation of the code they annotate.[6]
Java type annotations	Annotations that are applied to *type use*, which support improved analysis of Java programs way of ensuring stronger type checking.[7]
Pluggable type checker	A custom module used in conjunction with the Java compiler to ensure the state of a variable, with the goal of preventing or detecting all errors of a given variety.
Checker Framework	A pluggable type checker tool developed at The University of Washington[8] that comes with predefined type checkers and provides a framework for developers to build their own custom type checkers.

[2] http://www.cs.iastate.edu/~panini/about.shtml#what
[3] http://www.cs.iastate.edu/~panini/about.shtml#what
[4] http://www.cs.iastate.edu/~panini/about.shtml#capsule
[5] http://www.cs.iastate.edu/~panini/docs/faq.shtml#q_l_whatis
[6] http://docs.oracle.com/javase/tutorial/java/annotations/index.html
[7] http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html
[8] http://types.cs.washington.edu/checker-framework/