Summer UROP Proposal

Joshua Tomazin

Motivation and Problem Statement

Organizing software according to layers of abstraction and functionality is the standard way of managing complexity in software design. It is the onus of the programmer to architect and explicitly connect these layers. However, this connective code can clutter software. A programmer hoping to understand a piece of software must recognize the entry point of the program, and then explore the modules and sub-modules of the program by browsing the dependencies of each part. Although this process is not difficult when the software is well-structured and cleanly modularized, it can be very difficult when the software is not. If layers of abstraction were explicit in the programming paradigm, reading code would be substantially easier.

Additionally, programs are usually annotated in a number of ways: comments, types, compiler optimizations, etc. These annotations sometimes make code easier to understand, but can also clutter and obfuscate the purpose of the code. Suppose it were possible to toggle the visibility of these annotations such that either a high-level or a low-level view of the program was visible.

Layered programming is a programming paradigm that has been only barely explored, and could provide these benefits as well as others. Layered programming allows and encourages a programmer to separate her program into a number of layers that encapsulate various details, and allows her to read and write code in these different layers. Layered programming differs from the general organizational strategy discussed earlier by explicitly specifying the layers and allowing easy and intuitive navigation through them. In order to allow easy construction and viewing of these layers, a layered programming tool in a text editor or IDE is necessary.

I propose further research into layered programming to create a data structure that manages layers and their relationships to each other, and an Emacs tool that would leverage this data structure and provide layered programming capabilities to a programmer. This tool should be language-agnostic and leverage Emacs' existing understanding of the language to manipulate it.

Prior Work

I've done only relatively preliminary research into prior work, but I've found two substantial works that include ideas related to layered programming.

The first of these is a language-level implementation of layered programming called o:XML. o:XML is a programming language written in XML, and leverages the flexible structure of XML to allow detailed documentation and unit tests to reside alongside the code with which they're concerned. However, this implementation doesn't allow layers to build on top of each other, but instead just separates parts of the program in a structured way. Furthermore, o:XML is a fairly unpleasant language to read or write because of the highly decorated syntax.

The other is literate programming as proposed by Donald Knuth. In literate programming, the programmer describes her program in plain English, with source code interspersed throughout with references. A specialized compiler then recognizes the source code and the relationship between its various pieces and then assembles an executable program. This compiler can also extract human-readable documentation from the same written program. Literate programming is relevant to layered programming because the documentation and source code exist in separate layers, and are managed and manipulated by a literate programming tool. Furthermore, literate programming tools can be language-agnostic.

Implications

Aside from the organizational and legibility benefits of layered programming, there is another potentially enormously powerful benefit. By explicitly separating out things like type annotations, it is possible to add type checking to untyped languages and remove type checking from typed languages. If the data structure that tracks this information is sufficiently rich, it would be possible to transpile from any supported language to any other without most of the parsing into an intermediary representation as is normally necessary. This could allow a program to be written seamlessly in whichever language best suits a particular part of the program.

Plans & Schedule

Weeks 1 & 2

- Research prior and existing work
- Discuss ideas
- Develop plan for data structure and means to interact with it through Emacs

Weeks 3 & 4

- Learn Elisp
- Design Emacs tool user interaction, begin implementation
- Figure out representation of layers and their relationships

Weeks 5, 6, & 7

- Write tools for navigating through different layers
- Write representation of layers
- Write code connecting layer/display and management controls to underlying representation

Weeks 8, 9, & 10

- Modify layered programming tool as necessary

Create a typed Scheme using layered programming:

- Integrate Emacs syntax understanding into layered programming tool
- Construct a type management layer that sits between Emacs and the Scheme interpreter