Efficient and Change-Tolerant Serialization for Program Analysis Tool-Chains

Timm Felden
University of Stuttgart, Institute of Software Technology
Universitätsstr. 38, 70569 Stuttgart, Germany
feldentm@informatik.uni-stuttgart.de

Abstract

We present a novel approach to platform and language-independent tool chain creation and maintenance. The key strategy is to rely on an efficient general-purpose serialization mechanism.

1 Motivation

Software analysis tool chains create huge amounts of data that have to be stored between the various phases of an analysis session. It is often convenient to write different parts of the tool chain in different programming languages. For example, the front-end might already be available as a C++ tool. There might be some analyses available in Ada. It could be efficient to implement prototypic analyses in Haskell. It may be required to provide a scripting interface in Python and the user interface in Java. All of these choices might be obvious per se, but the resulting software stack is simply a nightmare.

This is a common problem especially in academia. Usually, there are many loosely related projects which try to interact but often fail to do so because there is no acceptable way of creating a sufficiently efficient interface between the different tools.

In order to solve this problem, we started developing the SKilL language\cite{1,2}, which strives to provide an easy-to-use serialization mechanism that is efficient, platform and language independent, and sufficiently change tolerant.

The project has its origin in the development of the Bauhaus tool chain\cite{4}, where providing a means of creating new tools in any programming language that is known by an arbitrary undergraduate student is a matter of survival.

2 Related Work

There is plenty of related work in the field of serialization. Important general-purpose alternatives are XML, JSON and IDL\cite{3}. Approaches to high performance serialization are Apache Thrift, domain or language-specific mechanisms as well as hand-crafted file formats. The inspiration for SKilL came from Java Bytecode, LLVM/IR and Bauhaus IML. The specification language is heavily influenced by Thrift, C++ and Java.

3 SKilL

The SKilL approach to serialization consists of four areas, namely specification, API, binary file format and glue code. The specification language describes user-defined types known by the target tool. There is a trade-off between a rich type system and portability. Our solution is to provide type-safe pointers, IEEE-754-floats, unicode strings, basic containers (like sets and maps), and single inheritance. Furthermore, there is a built-in extension point type that can be used to create safe references to arbitrary types, and an annotation system that can be used to restrict usage of some types, e.g. by restricting a float to values between 0 and 360 (see \cite[§3-5]{2}). The specification language is designed to be readable by C++ or Java programmers without any further instructions. An example is given in listing 1. More examples can be found in \cite{2}.

The specification is fed into a generator which creates a language-specific "binding". The binding consists of the API which provides a means of reading and writing files and accessing data and glue code to realize the API and the SKilL specification. A Java API, for example, would contain a Node interface with getter and setter methods for color and edges with respective types. Additionally, the API contains some access and maintenance functionality, such as iterators over instances of types and factory methods for new instances.

The generated glue code implements the API, en-
we write the nodes to disk and release them for garbage collection. The written file is read subsequently. After reading the file, another \( n \) nodes with the color "orange" is added with 100 edges pointing to random non-orange nodes. This change is appended to the existing file.

The graphs are created using a deterministic random number generator, in order to make results comparable. Figure 2 shows time spent after initial creation, write, read and append. Note that read has to allocate instances again. We can see that allocation time is dominating the total execution time. Figure 1 shows the average size of a serialized Node.

5 Future Work

The evaluation was done with a prototypic Scala binding. Initial implementations of generators for Java, C++ and Ada are expected to be available in the second half of the year. We hope to improve the speed of development by offering SKiL under an open source license and making it publicly available. We are also looking forward to research projects using SKiL.

A first revision of the language specification is expected to appear by the end of the year, fixing several minor errors and providing obvious improvements, such as a bitvector type and user-defined default values. Furthermore, there will be a reliable comparison of SKiL and other common serialization mechanisms.

References