Tracing Abstractions through Generation

Karl Trygve Kalleberg
Department of Informatics
University of Bergen
karltk@ii.uib.no

2004-06-29

1 Background

When combining abstraction mechanisms native to a programming language, one can usually rely on complete and sensible debugging support. A regular debugger for an imperative language knows about functions, records, unions, and so on. It is even the case that common mistakes are explained at compile-time. Examples of this is the Jikes compiler[1] for Java and the MIPSPro compiler for C++, which both emit detailed semantical errors and warnings.

As generative programming is frequently used to add new abstraction features to existing languages or generate large parts of the code from formal specifications, the language compiler's output often becomes less relevant: If the warning pertains to code inside a programmer-editable template, the programmer may fix the problem, but often the compiler warns about issues inside automatically generated code which the programmer has no way of changing, unless he is also a developer of the generative programming system itself.

2 Some common problems

The problem is frequently compounded by the following factors:

- *Code as text*: The generative system treats code as text, and operates entirely on strings.

  The appeal of this approach lies in its availibility (there are literally hundreds of text preprocessors) and general nature (not tied to any particular language).

  The string operations do not treat the text as structured, which frequently results in syntactical errors, such as a missing '}' to close a code block.

  Most pre-processor systems fall into this class, with Java ServerPages[2], PHP[3] and the C/C++ preprocessor being well-known examples.

- *Late semantical analysis*: The generative systems may treat the code as synactic trees, usually operating on a CST or an AST representation of the

underlying language. It is rare however, that the rewrite rules have access to full semantic information for the elements it rewrites, and static semantic analysis is often left as an excercise to the compiler. Notable exceptions are the Eclipse JDT 3.0 for Java[4] and CodeBoost for C++ [5],[6], which offer (close to) full semantic analysis to the rewrite rules.

- *Step-wise rewriting*: Generative techniques may often generate code that, when sent to the compiler, is very different from the individual components it was generated from, making traditional debugging difficult.

- *Dynamic generation and loading*: In some forms of generative programming, the assembled binary code is produced on-demand inside a running system, presenting opportunities for undiscovered bugs, even syntactic ones, after customer deployment time. These flaws are supposed to be caught by various coverage analyses, but experience shows that this is difficult in practice.

The problem of abstraction traceability has been known since the inception of compilers, and is also a big issue in world of modelling these days. For both imperative and object-oriented languages, annotating the produced assembly code with line number (and source code excerpts) proved effective, and has become a de facto part of most compilers.

The annotation solution cannot easily be applied to generative programming in the general case, however, as the coupling between component (or template) source code and the final, generated product is extremely loose. What appears as one line in one component may end up at hundreds of places after generation.

This problem bites most forms of code generation and compositions, in particular implementors of domain-specific languages (DSLs) as well as aspect-oriented programming.

3 Some suggested solutions

In addition to experience from embedded DSLs in languages with extensive metaprogramming capabilities, such as Lisp, we have:

- *Annotations coupled with tool support* If annotations are available for inspection by tools inside the final product, inspectors can be written that relate generated code to initial components and specifications. This has some relation to debug slicing [7].

- *Rigorous synactical and semantical analysis at rewriting time* Provided with full-fledged semantic analysis, and possibly domain-specific rules for the particular components which encode their semantics, errors would be caught at earlier stages, where the distance between the initial components and the final source code is smaller.

- *Step-wise rewriting with inspection* Interactive playback of the rewrites may also be helpful to track the origins of particular problem areas.

References

[1] Jikes Java Compiler . http://www.research.ibm.com/jikes/, 2004.

[2] Java ServerPages. http://java.sun.com/products/jsp/, 2004.

[3] PHP. http://www.php.net/, 2004.

[4] Eclipse Team. eclipse.org, 2004.

[5] Otto Skrove Bagge. CodeBoost: A Framework for Transforming C++ Programs. Master's thesis, University of Bergen, Norway, 2003.

[6] Karl Trygve Kalleberg. User-configurable, high-level transformations with CodeBoost. Master's thesis, University of Bergen, Norway, 2003.

[7] István Forgács Tibor Gyimóthy, ÁrpÁd Beszédes. An efficient relevant slicing method for debugging. In M. Lemoine O. Nierstrasz, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science, Toulouse, France, September 1999. Springer-Verlag.