Sometimes I want to run a garbage collector on the internet.

– Eelco Dolstra

12 Discussion

This chapter discusses different approaches for achieving language independent transformations and some fundamental tradeoffs related to these. The chapter also briefly discusses the relation of program transformation to other approaches for software evolution. Additionally, it contains a discussion on the place for open research systems in the pursuit of better practical transformation techniques.

12.1 Techniques for Language Independence

This dissertation is concerned with the development of techniques for expressing language-independent program transformations. Its motivation has been to find ways of making transformations applicable across different subject languages quickly and easily. The techniques proposed herein are by no means exhaustive. A discussion of alternative approaches is therefore warranted.

There are several possible directions for achieving language-independent program transformations. All of them must tackle the tradeoff between two opposing requirements: the need to hide language details versus the level of detail required by a given transformation. Abstracting over language details hides irrelevant differences in subject languages and enables higher-level program models. Consequently, transformations written for these models may be applied to languages which are abstractable into the higher-level models. The requirements placed on the model varies greatly between transformation tasks, however. The contents of a model used to analyse the module dependency graph of a program is quite different from one which is used for intra-procedural control-flow analysis. The combination of several approaches is therefore more likely to give good results.

12.1.1 Abstracting over Data

Capturing software in a high-level and general program object model (POM) is a data-centric approach to language independence. The goal of this technique is to abstract over irrelevant language details and provide a uniform model across subject languages. Abstract syntax trees (ASTs) may be seen as the first step in this direction. ASTs abstracts over most of the syntactical "noise" in the subject language, but may still be used to reproduce the original program faithfully modulo layout and comments. POMs may be arbitrarily more abstract. The PROGRES example in Chapter 2, Section 2.4.2 illustrates how the concept of program configurations may be described abstractly. This abstract model could support transformations, but mapping these transformations from the abstract model back to equivalent operations on the original source code is generally a hard problem to solve.

The fundamental tradeoff for the data-centric technique is the choice of what should be considered irrelevant language details. This clearly depends on the transformation problem for which the abstract model should be used. Arriving at a final authoritative language independent program object model is therefore very likely to be infeasible. A more versatile approach is required which can account for the varied needs of the transformations.

12.1.2 Expressing Generic Algorithms

Formulating generic and parametrised transformation algorithms, where language specific components can be inserted, may be considered a "function-centric" approach to language-independence. The strategic programming paradigm supports this approach well by dividing transformation programs into general transformation logic and data processing rules. The principle is that the general transformation logic, in the form of strategies, is designed for the application of language-specific data processing rules. By replacing the rules, the same logic may be applied to different subject languages.

This approach works well for a good number of problems, but suffers from drawbacks discussed in Chapter 5, Section 5.4.3. An additional drawback, particular to program transformation, is that the designer has no abstract signature to write the algorithm against. The model abstracting over subject languages – what may be called an abstract language – is never defined explicitly in the transformation program. Instead, the algorithm is written against a mental model hidden in the rule set. The model is purely conceptual, but the transformation must nonetheless respect it. The lack of a clear abstract language definition often makes it very hard to reason about the transformation. At best, the model exists in the form of well-written documentation. Accidental violation of the model is easy because the transformation language compiler cannot check any of its rules. In the course of development, the transformations are often tested against a concrete selection of subject languages to raise confidence in their correct behaviour. This may easily introduce an unintended bias in the formulation favouring the example selection.

Describing abstract languages for language-independent program transformation is still an open research problem. It is possible that the answer may be found from studying better and more flexible ways of describing computer languages in general.

12.1.3 Adapting Generic Algorithms

Aspects may further improve the genericity of an algorithm by exposing additional variation points for the algorithm user. In some cases, accidental implementation bias may be corrected without changing the algorithm.

12.1.4 Modular Language Descriptions

Finding a good formalism for describing computer languages in small, reusable modules has been and, many would claim, still is a long-standing goal of computer science. The approach taken in this dissertation is to describe the language semantics as transformation libraries that define how the various languages features are translated into a minimal core language and, eventually, (via a machine-specific compiler or interpreter) into executable machine code. A clear drawback of this approach is that the description is very tied to its eventual application: the compilation and execution of programs. Unless special care is taken during the design of the transformation library, reusing the implementation for other tasks, perhaps for program validation or defect checking, may be impossible. Research into modular language semantics holds some hope that "problem neutral" descriptions may eventually be possible.

The research towards modular language descriptions [Mos04a] is concerned with capturing the semantics of programming languages into small modules that each describe a particular language feature. These modules may be composed with other modules. The final composition describes the entire language and may form the basis of any language processing tool for the composed language such as a compiler, type checking front-end, refactorer or defect checker.

Various formalisms for defining modular semantics have been devised including monadic denotational semantics [Mog91], abstract state machine montages [KP97], action semantics [DM03] and modular structural operational semantics [Mos04b]. None of these formalisms have seen significant adoption, unfortunately.

A firm and general basis for describing subject language semantics could be of great use for transformation developers. For example, determining whether it is possible to find a reasonable mapping from a given subject language to a given abstract language could become easier. Additionally, it could improve the means available for validating (or verifying) that a given transformation respects certain semantics of a subject language.

12.2 Other Approaches to Software Evolution

Program transformation is not the only component in a solution toward good software evolution. Therefore, it is important that the techniques proposed in this dissertation for expressing reusable, language-independent program transformations work well with the (relatively) new and forthcoming programming and software evolution methodologies such as refactoring, generative programming, interactive development environments, extensible languages and aspect-oriented programming. The experimentation performed during the development of the case studies in Part V shows that the techniques apply well to generative programming and that integration into interactive development environments is significantly easier due to the POM adapter. An early prototype for refactoring of Java code has been constructed. This suggests that implementing refactorings in a "strategic" style using the abstractions proposed in this dissertation may be very powerful. Additional experiments are necessary in order to gain more experience before a final conclusion can be made. Experience from the development of the domain-specific aspect language described in Chapter 8 suggests that aspect-oriented subject languages are reasonably well supported. Extensible languages are discussed in Chapter 13.

12.3 Availability of Research Systems

The investigation and analysis leading to the survey of software transformation systems in Chapter 2 uncovered that practically all of the (still active) research systems described in the literature are freely available for download. With only a handful of exceptions, the full source code for these systems were also provided. In many cases, the availability of source code proved crucial to understanding the detailed workings of many features. The availability of source code was also important during the analysis leading up to the POM adapter (Chapter 4). During this analysis, it became necessary to consult concrete compilers and transformation systems to account for design decisions usually glossed over in the literature.

An important theme of the dissertation is to demonstrate that the techniques proposed herein are applicable in practise. For this reason, all the source code for the software constructed for this dissertation, including the case study prototypes, is available for download via `www.spoofax.org`[1].

[1]The name "Spoofax" was selected because available `.org` domains are hard to come by.