Electronic Communications of the EASST
Volume X (2014)

Proceedings of the Second International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2014)

Managing Language Variability in Source-to-Source Compilers by Transforming Illusionary Syntax

Tero Hasu

4 pages

Guest Editors: Anya Helene Bagge, Vadim Zaytsev
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Managing Language Variability in Source-to-Source Compilers by Transforming Illusionary Syntax

Tero Hasu*

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
tero@ii.uib.no

Abstract:

A programming language source-to-source compiler with human-readable output likely operates on a somewhat source and target language specific program object model. A lot of implementation investment may be tied to transformation code written against the specific model. Yet, both the source and target languages typically evolve over time, and compilers may additionally support user-specified, domain-specific language customization. Language workbenches commonly support generating object model implementations based on grammar or data type definitions, and programming of traversals in generic ways. Could more be done to declaratively specify abstractions for insulating the more language-semantic transformations against changes to source and target language syntax? For example, the idea of views enables pattern matching and abstract data types to coexist—could similar abstractions be made pervasive in a generated program object model?

Keywords: Language adaptation, program representation, Racket, transcompilers

1 Introduction

A programming language implemented as a compiler generating source code allows for reuse of existing infrastructure for the target language. Such a language can also enable abstraction over target language versions, implementations, and idioms (such cross-cutting concerns can be particularly pressing in a cross-platform setting). If the source-code-generating compiler furthermore produces human-readable, high-abstraction-level output, then it also has a low adoption barrier in the sense that it can be regarded merely as tools assistance for programming in the target language. We use the term *source-to-source compiler* (or *transcompiler* for short) for such language implementations.[1]

A transcompiler typically translates its source language into its target language through successive program transformation steps. Each transformation step is programmed against a *program object model* (POM), which includes at least a data structure used to represent a program,

* This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

[1] An established definition for the term "source-to-source compiler" encompasses any compiler that produces its output *in* a high-level language, even when the output itself is low-level enough to read like assembly. For lack of a dedicated term for our more narrow definition, we simply use "source-to-source compiler" in the more narrow sense.

and a programming *interface* (or API) for manipulating the data. The POM (or POMs) used should be able to represent both source and target language programs, and any in-between language. It is common to define an intermediate language (or a *core language*) with a simpler, somewhat language agnostic syntax, which is something in between "desugared" source language and "ensugared" target language.

A compiler codebase mostly transforming core language may have fewer conditional cases (due to having fewer language constructs to manipulate) and less dependency on the *subject language* (i.e., source language, target language, or both [Kal07]). Still, as a source-to-source compiler's output should retain a high level of abstraction, sufficient semantic information must be carried through the compilation pipeline to allow high-level constructs to be preserved or recreated. This requirement for a "wider communication path" between the compiler front and back ends makes it hard to avoid language specificity. Having a large part of a compiler's codebase tied to language specifics may make it costly to maintain as languages evolve.

Languages may also change not due to design changes in the course of their evolution, but due to being designed to be extensible or otherwise adaptable to domain-specific purposes. It may be desirable to extend a transcompiler's source language with relevant abstractions for programming application or platform specific components. Likewise, it may be desirable to customize the output for different program configurations, perhaps to pick a supported or customary error handling mechanism [Has12], for instance. Where the extension mechanism is merely capable of mapping extended source language to unextended source language (e.g., traditional Lisp macro systems), an extension has no impact on the compiler. However, there also are extension mechanisms powerful enough to enable domain-specific optimizations [RSL+11, TSC+11], for example, and indeed there might be several extension points within a compilation pipeline [Bag10].

While there are promising approaches (e.g., based on attribute grammars [SKV13] and modern object-oriented language features [HORM08]) to enable modular language extension, it is as yet unclear just what kind of language adaptation is possible without having to manually adjust the compiler codebase. For now, to help achieve some degree of insulation against language variability in transcompiler engineering, we advocate working on tools support that encourages the use of abstraction (over specific language constructs) in programming code transformations. Although there are solutions for adapting existing POMs for existing interfaces (e.g., Kalleberg's POM adapter technique [Kal07]) and extending existing transformations to meet new requirements (e.g., through *aspects* [Kal07]), there is more work to be done in the area of providing a richer variety of APIs to program transformations against in the first place.

Structural abstraction is already widely supported in the form of generic traversals of some kind (e.g., term rewriting strategies, visitor design pattern). Stratego, for example, provides primitive traversal operators (**one**, **some**, and **all** generic functions) for all abstract syntax tree node types, as well as combinators for specifying more complex traversals. We are now looking for better support for language-semantic abstraction. This might entail allowing the programmer to declaratively express commonalities and relationships between syntactic constructs, and having the language development toolkit then readily offer support for writing transformations that avoid needless syntax specifics (where only a more general characteristic is of interest).

There are a number of existing tools (e.g., GOM [Rei07] and ApiGen) capable of generating a program model from a language grammar description (or similar), but the generated API invariably just follows the structure of the grammar. Thus, while it may be possible to declare some

```
(struct DeclVar (id t)) ;; variable binding (declaration)
(struct Var (id)) ;; variable reference (value expression)
(struct NameT (id)) ;; type name reference (type expression)
```

Figure 1: Grammatically unrelated syntax objects types (defined as Racket structures) with a commonality: each of them contains an identifier. An interface capturing the commonality might include a predicate, an id symbol accessor and mutator, etc. A global renaming implemented in terms of such an abstraction should be fairly decoupled from the underlying grammar.

```
(struct DeclVar (id t))
(struct DefVar (id t v))
```

```
(struct DefVar (id t v))
(struct Undefined ())
```

```
(struct DeclVar (id t))
(struct DefVar DeclVar (v))
```

Figure 2: Alternative declarations of syntax object types for the same abstract syntax: (1) unrelated types; (2) DeclVar is DefVar with Undefined initializer expression; and (3) with subtyping, variable definition being a special case of variable declaration.

commonalities between constructs deriving from the same non-terminal, other commonalities (such as show in Figure 1) typically require ad hoc, handwritten code to capture. We might want to augment a POM generator to implement interfaces that are independent of both subject language grammar and POM data representation. Such interfaces might span multiple (or no) object types, only provide access to certain portions of available information (possibly in a variety of formats), and yet appear similar to actual syntax objects' interfaces. It is unclear as to what kind of commonalities could conveniently be declared for purposes of code generation, and how.

An abstraction-friendly POM generator might benefit a compiler engineer by: making it less effort to implement multiple interfaces to choose from on a case-by-case basis (i.e., whichever seems most convenient for a given transformation); limiting breakage of abstraction-using transformation code upon subject language modification; and providing some insulation against incidental, implementation-specific grammar or program representation choices (such as shown in Figure 2). The possibility of effortlessly exposing various interfaces should make picking the "best" concrete choice less crucial in the first place. Even the distinction between object fields and *annotations* (i.e., open-ended collections of secondary, "non-structural" information in syntax objects) should become less prominent.

Pattern matching is commonly used in program transformations. Programming against abstract interfaces need not always mean the loss of pattern matching. The Tom system, for example, abstracts over concrete data structures by allowing rewriting based on algebraic terms that map to actual data structures [Rei07]. For further abstraction one might—following the philosophy of *views* [Wad87]—(seemingly) expose as many "data representations" per program object as desired. Some languages have sufficient "hooks" to enable "abstract algebraic views" to be implemented for purposes of pattern matching (e.g., as demonstrated in Figure 3).

A drawback of the kind of abstraction we have sketched is that it precludes the use of concrete syntax in patterns and templates, as supported by some language workbenches (e.g., Rascal and Spoofax). The illusion of semantic commonality capturing interfaces being like those of actual syntax objects can also be incomplete in respect to other abstraction mechanisms. Suppose a

```
(define-match-expander DeVar
  (syntax-rules ()
    [(_ id t) (or (DeclVar id t) (DefVar id t _))]))
```

```
(match (DefVar 'x 'T (Literal 5))
  [(DeVar id t) (list id t)])
;;=> (x T)
```

Figure 3: Defining and using a custom pattern matching form DeVar in Racket, for matching both DeclVar and DefVar objects and their relevant fields. This implementation of DeVar works for representations (1) and (3) in Figure 2.

particular algebraic view includes a specific annotation (thus making it look structural), but a generic traversal does *not* traverse any annotations; now there is a discrepancy, but it is not clear how the behavior of traversals should be affected by having multiple interfaces per object that may be used to query for traversable sub-objects.

Bibliography

[Bag10] A. H. Bagge. Yet Another Language Extension Scheme. In Brand et al. (eds.), *SLE '09: Proceedings of the Second International Conference on Software Language Engineering*. LNCS 5969, pp. 123–132. Springer-Verlag, Mar. 2010.

[Has12] T. Hasu. Concrete Error Handling Mechanisms Should Be Configurable. In *Proceedings of the 5th International Workshop on Exception Handling (WEH'12)*. Pp. 46–48. IEEE, June 2012.

[HORM08] C. Hofer, K. Ostermann, T. Rendel, A. Moors. Polymorphic Embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08, pp. 137–148. 2008.

[Kal07] K. T. Kalleberg. *Abstractions for Language-Independent Program Transformations*. PhD thesis, University of Bergen, Norway, Postboks 7800, 5020 Bergen, Norway, June 2007. ISBN 978-82-308-0441-4.

[Rei07] A. Reilles. Canonical Abstract Syntax Trees. *Electron. Notes Theor. Comput. Sci.* 176(4):165–179, July 2007.

[RSL+11] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, K. Olukotun. Building-Blocks for Performance Oriented DSLs. *ArXiv e-prints*, Sept. 2011.

[SKV13] A. M. Sloane, L. C. L. Kats, E. Visser. A Pure Embedding of Attribute Grammars. *Sci. Comput. Program.* 78(10):1752–1769, Oct. 2013.

[TSC+11] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen. Languages as libraries. *SIGPLAN Not.* 47(6):132–141, June 2011.

[Wad87] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87, pp. 307–313. 1987.