Exploiting Macros in Source-to-Source Compiler Implementation *

Tero Hasu

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

Abstract

A sufficiently feature-rich general-purpose programming language with an expressive macro system can play multiple roles in the implementation of a source-to-source compiler: it can host the language being compiled, and expose its own macro system to make the hosted language user extensible; it can embed macro-implemented language for the domain of program transformations; and its general-purpose features and libraries allow for the entire compiler to be implemented based on the same language technology. I discuss some potential uses of the "programmable programming language" Racket in the implementation of source-to-source compilers.

1 Introduction

A *source-to-source compiler* (or *transcompiler* for short) is a programming language implementation outputting source code. Transcompiled languages can be useful in reusing target language infrastructure and abstracting over target language variability, and—particularly when translated into human-readable code—their adoption need not entail high risk [3].

Transcompilers and Lisp-style macro expanders are conceptually similar in that they translate between languages, and usually operate on abstract syntax. This suggests that a macro system might be used to do some of the work of a transcompiler, particularly on the front end side, which is where macros are designed to operate. The Racket programming language [2] has a particularly expressive macro system, and it has also been designed to host other languages defined as Racket libraries [8]. With some language design compromises for Racket compatibility, a Racket-hosted language may get significant reuse out of the host language facilities. These, in turn, can serve as a convenient basis e.g. for surface syntax implementation, desugaring transformations, and an extension mechanism for the hosted language.

There are also many potential uses for *domain-specific languages* (DSLs) in the program transformation domain, as suggested by the Spoofax language workbench [5] for instance, with its selection of DSLs for defining various aspects of language implementations and their tools support. In transcompiler implementations DSLs are commonplace particularly in the specification of parsers and data structures for representing programs being transformed. While a Racket-hosted language can get parsing almost "for free", a *program object model* (POM) is still a likely implementation requirement. A POM includes at least a data structure used to represent a program, and a programming interface (or API) for manipulating the data.

In the case of a transcompiler, an *abstract syntax tree* (AST) would typically be a useful POM structure. As each syntactic construct in the transformed language usually gets its own AST

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

node data type, an AST implementation tends to involve repetitive code. There are a number of existing tools (e.g., ApiGen [1] and GOM [6]) capable of generating syntax tree definitions from a language grammar description (or similar). A language like Racket is also capable of performing the required code generation with macros, thus enabling declarative programming of ASTs within the language. One might even consider making it possible to declaratively specify abstractions that are not restricted to grammar structure, but rather reflect some other structural or conceptual similarity in the constructs of the transformed language [3].

A source-to-source compiler implementation language equipped with a sufficiently expressive macro system makes it possible to get the convenience of (program transformation) domain specific language without losing the flexibility of a general-purpose programming language. Macro-enabled language malleability also opens up opportunities for hosting the implemented language on top of the implementation language, which allows for varying degrees of sharing of language infrastructure between the two languages; the achievable level of sharing depends on the used language integration approach [4, 7], as well as the design of the hosted language.

References

[1] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59, April 2004.

[2] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[3] Tero Hasu. Managing language variability in source-to-source compilers by transforming illusionary syntax. In *Proceedings of the 2nd International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2014)*, pages 11–14, February 2014.

[4] Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. Implementation of S-expression based extended languages in Lisp. In *Proceedings of International Lisp Conference*, pages 179–188, Stanford, California, USA, June 2005.

[5] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[6] Antoine Reilles. Canonical abstract syntax trees. *Electron. Notes Theor. Comput. Sci.*, 176(4):165–179, July 2007.

[7] Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *European Lisp Symposium*, May 2014.

[8] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *SIGPLAN Not.*, 47(6):132–141, June 2011.