Incremental Parametric Syntax for Multi-Language Transformation

James Koppel
MIT
Cambridge, MA, USA
jkoppel@mit.edu

Armando Solar-Lezama
MIT
Cambridge, MA, USA
asolar@csail.mit.edu

Abstract

We present a new approach for building source-to-source transformations that can run on multiple programming languages, based on a new way of representing programs called **incremental parametric syntax**. We implement this approach in our Cubix system, and construct incremental parametric syntaxes for C, Java, JavaScript, Lua, and Python, demonstrating three multi-language program transformations that can run on all of them. Our evaluation shows that (1) once a transformation is written, relatively little work is required to configure it for a new language (2) transformations built this way output readable code which preserve the structure of the original, according to participants in our human study, and (3) despite dealing with many languages, our transformations can still handle language corner-cases, and pass 90% of compiler test suites.

CCS Concepts • **Software and its engineering → Translator writing systems and compiler generators**; *Syntax*; **General programming languages**;

Keywords abstract syntax trees, program transformation

ACM Reference Format:
James Koppel and Armando Solar-Lezama. 2017. Incremental Parametric Syntax for Multi-Language Transformation. In *Proceedings of 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3135932.3135940

1 Introduction

As the scale of software grows, developers will increasingly depend on program transformation tools to help maintain software. Programmers use transformation tools to do everything from small-scale refactoring to modernizing entire legacy applications. Most tools are wedded to one language (or even one compiler), and often require hundreds of thousands of lines of code to implement. Given that developers use hundreds of languages, the value of these tools would be significantly enhanced if they could be easily made to work across multiple languages.

Many program analysis frameworks have addressed the multi-language problem by reducing multiple languages into a common *intermediate representation*. This approach fails when the goal is *program transformation*, because translating into an intermediate representation necessarily destroys information. An alternative to reduction is to define languages in a modular way that allows representations of different languages to share common structures. In the last few decades, researchers have proposed a number of techniques to deal with different languages modularly, including work on modular semantics [2], modular interpreters [4], and modular syntax [1]. The limitation is that using any of these techniques requires defining an entire language in a specialized manner, differently from existing tools. Hence, so far they have only been applied to small languages such as DSLs.

We present **incremental parametric syntax**, which allows implementers to define languages modularly on a much greater scale than previously, and hence write source-to-source transformations that run on multiple real languages. We implement this approach in our Cubix system, and demonstrate it with several characteristic program transformations that each can run on several of C, Java, JavaScript, Lua, and Python. We show that developers can define these transformations in a few lines of code, and that the output of the tool does not suffer from the readability problems that plague IR-based approaches. In fact, we conducted a human study that showed that the output of our tool is no less readable than code that was transformed by hand.

We give a brief overview of this work in the remainder of this paper. More details are available in the full paper [3].

2 Approach

2.1 Incremental Parametric Syntax

Our approach instead is based on the idea of *incremental parametric syntax*. Conceptually, one can think of our approach as offering the following functions:

SPLASH Companion'17, October 22–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5514-8/17/10…$15.00
https://doi.org/10.1145/3135932.3135940

```
data Arith =
    Add Atom Atom
data Atom
    = Var String
    | Const Lit
data Lit
    = Lit Int
```

```
data ArithL; data AtomL; data LitL
data Arith t | where
  Add :: t AtomL →t AtomL
                 → Arith t ArithL
data Atom t | where
  Var :: String → Atom t AtomL
  Const :: t LitL → Atom t AtomL
data Lit (t :: * → *) | where
  Lit :: Int → Lit t LitL
```

Figure 1

```
decompose_J :: Java →IR ⋈ Rem_J, decompose_C :: C →IR ⋈Rem_C
transform   :: ∀x . IR ⋈ x → IR ⋈ x
recompose_J :: IR ⋈Rem_J →Java, recompose_C :: IR ⋈Rem_C →C
```

This decomposes a language into generic and language-specific fragments. A transformation can then be defined only on the generic fragments. This allows the transformation to run on any language that contains the generic, while still preserving information, and leaving the language-specific parts untouched. This approach is incremental, in that a programmer can translate language-specific fragments of a language into generic ones in a piecewise fashion.

The composition $X \bowtie Y$ is done by the "data types à la carte" approach [5]. To that we add **sort injections**, which solve the problem that, when two languages have similar nodes such as assignment nodes, they may interact differently with the rest of the syntax. A sort injection is a node or sequence of nodes which allows terms of one sort to be used at another sort. For instance, they allow one to specify that a generic assignment may be used where a language-specific statement is expected, and to independently specify what terms may appear as the LHS and RHS of an assignment. Using these sort injections, Cubix can generate code for a new representation of the language which is isomorphic to the original one, and which allows intermixing language-specific and generic fragments.

2.2 Automatically Generating a Modularized Representation

Generating an incremental parametric syntax for a language requires that the language first be decomposed into fragments. The comptrans tool can automatically generate such a decomposition from a pre-existing third-party syntax definition. It takes as input a syntax definition as a system of mutually recursive algebraic datatypes, and outputs code for a syntax definition as a compositional data type [1] isomorphic to the original. An example is given in Figure 1. The LHS gives a representation of an abstract syntax as an ADT. comptrans transforms this input into the GADTs on the right.

3 Results

We have implemented our approach in the Cubix system. Cubix is organized as a collection of libraries which users can use to assist in building incremental parametric syntaxes and multi-language transformations. We built support for C, Java, JavaScript, Lua, and Python, and built three example program transformations: hoisting, test-coverage instrumentation, and conversion to three-address code (TAC).

We ran our transformations on language test suites for each of the 5 languages, and checked whether they still passed. Our results are promising, showing pass rates of 98.2%, 97.8%, and 95.8% for the Hoist, Testcov, and TAC transformations respectively. One caveat is that, for many tests, performing the identity transformation, which reformats the code, would cause the test to fail. This could happen both because of bugs in the third-party pretty-printers, or because the test is self-referential. These numbers exclude tests which fail the identity transformation.

3.1 Readability: Human Study

We ran a study to evaluate the readability of our transformations' output. First, we ask a set of human contributors to transform programs by hand. We then give a separate set of human judges from Mechanical Turk these programs, alongside the corresponding automatically transformed programs, and ask them to rate them both on correctness and quality. We automatically reformat the human-written code before presenting them for comparison.

For the set of target programs in this study, we created a benchmark set called the RWUS suite (Real World, Unchanged Semantics), consisting of 10 functions of between 5 and 50 lines randomly selected from top Github projects. For each function, we created stubs so that it could be tested standalone, and created a thorough set of tests (7000 lines total) that could detect any change to the program semantics.

The Mechanical Turk judges were asked to rate pairs of programs on a 1-5 scale. The hypothesis for each language was that the automatically-transformed programs would receive an average rating of at most 1 less than the human-transformed. This is a problem in statistics known as non-inferiority testing. We tested this with a paired t-test, obtaining that the hypothesis held for each language with $p < 10^{-9}$.

Acknowledgments

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

References

[1] Patrick Bahr and Tom Hvitved. 2011. Compositional Data Types. In *ICFP*.

[2] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. 2013. Modular Monadic Meta-Theory. In *ICFP*.

[3] J. Koppel and A. Solar-Lezama. 2017. Incremental Parametric Syntax for Multi-Language Transformation. *ArXiv e-prints* (July 2017). arXiv:cs.PL/1707.04600

[4] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*.

[5] Wouter Swierstra. 2008. Data types à la Carte. *Journal of functional programming* 18, 04 (2008), 423–436.