Synthesising a Compiler

Josh Reese

jreese@cs.umd.edu

I. Introduction

Reasoning about the functionality and properties of real programming languages is a challenging and arduous task. In fact there are several groups working on formally specifying the semantics of JavaScript for various reasons [7, 4]. In some cases it is useful to have a core language with well defined semantics to help reason about the semantics of a more complicated, but related, language. However, translating from one language to a simpler core is no trivial task.

Recent work in program synthesis shows very interesting results for generating code that is able to fulfil previously defined specifications [6, 11, 2, 9, 10]. In this work we hope to show that given the specifications for a compiler we will be able to automatically generate code to translate from an expressive, rich language with many features (i.e. lots of 'syntactic sugar' to a much more simple language which can be easily reasonable about semantically.

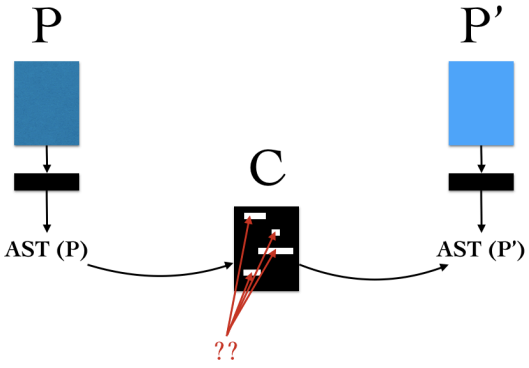


Figure 1: *Overview of translation of program P in language L to program P' in language L'*

II. Methods

In order to complete this work we will develop a system in Racket [1] which will synthesise a compiler from some language L toa language L' where $L' \subseteq L$. In order to achieve this goal we will need to first identify target languages to use for compilation. One potential target for the language L' is to use something akin to A-normal form (A-NF) [8] which will allow us to use a functional language as L that we can then transform to A-NF. In order to accomplish this we will be working with ASTs which represent programs in L and L'. Once we have trees that represent programs in our languages we can begin to synthesise transformations from trees of type L to trees of type L'. There has been extensive work on tree transformations [5, 3] which we will hopefully be able to draw from in order to automatically generate transformations.

The transformations we will be using are:

- Delete node
- Replace node
- Swap nodes
- Rotate (sub)tree
- TODO: more?

Our initial synthesis approach will be a brute-force search where we apply all the transformations in various orders until a correct transformation is found (or all possibilities are exhausted). In the case of attempting all transformations with no successful result we can conclude there is no possible transformation between these trees.

While this is certainly one way to approach this problem there has been work on synthesis that tries to evaluate the 'correctness' of an attempted solution using various criteria [13, 11]. Building on these ideas we could apply some transformation(s) to the ASTs, evaluate whether or not this was a 'good' transformation and use that to guide our searches.

A final possibility for guiding our searches is to perform a symbolic search on the trees to help synthesise the appropriate transformations. This will follow some of the ideas presented in Rosette [12]. This work is a framework for designing languages which use SMT solvers. In addition to this it is embedded in Racket which

contributes to our decision for using Racket in our work.
A pictorial overview of this can be found in Figure 1 and a sample de-sugaring can be found in Listing 1.

Listing 1: *"De-sugaring Scheme's cond macro"*

```
(cond cond-clause ...)
cond-clause =
      [test-expr then-body ...+]
    | [else then-body ...+]

(cond-desug desug-clause ...)
desug-clause =
      [if test-expr (begin then-expr ...+)
        #<void>]
    | [if #t (begin then-expr ...+)
        #void]
```

III. Timeline

In order to achieve the goals of this project we will need to complete a series of tasks which involve: continued background research, developing language models, developing a compiler for our languages, poking holes in this compiler, synthesising these holes, and evaluating our work. We have already been looking at papers related to program synthesis, and will continue to do so, but we will need to focus more on the specifics of the languages we are targeting. Specifically what 'sugared' syntax's we are interested in 'de-sugaring'. We will also need to determine a set of candidate tree transformations that can be used in our synthesis. Once there is some understanding regarding these concepts we can move on to developing the language models in racket that we will be using. Following this we can write the compiler for the language with holes in it. The real challenge to the project comes next: synthesising the transformations. This presents a number of challenges. One of which is gathering input data to be used during synthesis. Since we will be synthesising a tool which produces programs we will need to provide correct programs as inputs (in language L) and their corresponding correct programs as outputs (in language L'). Once this is finished we can move on to evaluating our work and preparing the necessary reports. A Gantt chart of the proposed time-line can be found in Figure 2.

IV. Evaluation

Evaluation of the final product will fairly straightforward. In order to test this we will write programs in language L, use our synthesised compiler to translate them to language L' and finally verify their outputs are equivalent. If this is the case we have succeeded. However, there are some subtle evaluation metrics that are important to consider as well. These include: the time it takes to synthesise a compiler for the language, how many language features the synthesised compiler can handle, the quality of code produced among others. All of these will need to be taken into account during evaluation. Additionally, as previously mentioned, generating test inputs for the synthesis process will prove a challenging task to overcome.

References

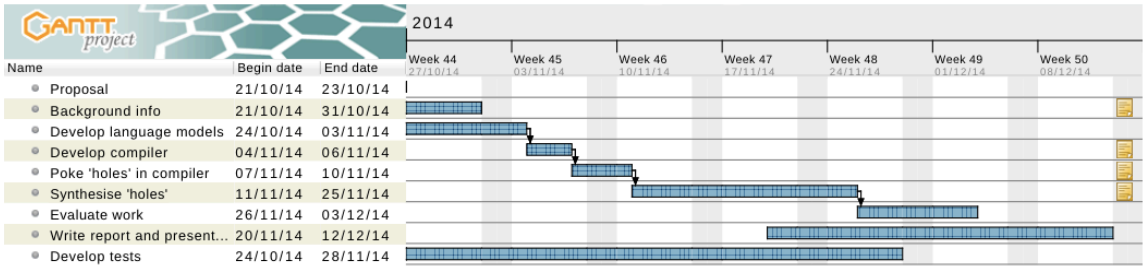
[1] The racket programming language. `racket-lang.org`.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2013.

[3] Loris D'Antoni, Margus Veanes, Benjamin Livshits, and Davi d Molnar. Fast: A transducer-based language for tree manipulation. Technical Report MSR-TR-2013-121, November 2013.

[4] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 126–150, 2010.

[5] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek.



(a)

Tasks 2

Name	Begin date	End date
Proposal	21/10/14	23/10/14
Background info	21/10/14	31/10/14
Exampes of de-sugaring to A-NF *Reading papers* *Learning Racket* *Ideas of tree transformations* *Examples of trees*		
Develop language models	24/10/14	03/11/14
Develop compiler	04/11/14	06/11/14
Extract tree operations for synthesis		
Poke 'holes' in compiler	07/11/14	10/11/14
Find places where we can synthesise transformations		
Synthesise 'holes'	11/11/14	25/11/14
Somehow synthesise transformations automatically. *Verify correctness of synthesis.*		
Evaluate work	26/11/14	03/12/14
Write report and presentation	20/11/14	12/12/14
Develop tests	24/10/14	28/11/14

(b)

Figure 2: *Figure 2(a) shows the time-line of the project while Figure 2(b) shows details about the tasks.*

[6] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 55, 2014.

[7] Sergio Maffeis, JohnC. Mitchell, and Ankur Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer Berlin Heidelberg, 2008.

[8] Amr Sabry Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Lisp and Symbolic Computation*, pages 288–298.

[9] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[10] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *IN 12TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS (ASPLOS 2006*, pages 404–415. ACM Press, 2006.

[11] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI '11, pages 492–503, New York, NY, USA, 2011. ACM.

[12] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152, 2013.

[13] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.