Synthesising a Compiler

Josh Reese

jreese@cs.umd.edu

I. Motivation

Reasoning about the functionality and properties of modern programming languages is a challenging and arduous task. However, in order to make any sense of programs it must be the case that the languages used can be analysed. One way to reduce the difficulty of reasoning about a language is to reduce the language itself to a much simpler core language. Examples of attempts to formalise Javascript can be found in [4, 6]. Having this core language with well defined semantics makes it much easier to 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 [5, 9, 3, 7, 8]. In this work we will leverage the ideas from program synthesis to automatically generate code to translate from the more complex, feature-rich language (which will call language F), to the simpler core language (language S). To accomplish this we will use specifications of the two languages and test inputs written in language F to synthesise a compiler from language F to language S.

II. Summary

It is the main goal of this work to automatically generate code that compiles programs written in some language, F, to some other language, S, where $S \subset F$. To do this we have implemented two separate languages in Rascal, interpreters for each, and a synthesiser, also in Rascal. This synthesiser takes, as input, programs written in language F and produces code which will translate programs in language F to programs in language S.

I. Rascal

Rascal is a metaprogramming language being developed primarily by the SWAT group at Centrum Wiskunde and Informatica in Amsterdam [2]. The motivation behind Rascal is to provide a single tool for effectively analysing, transforming and generating source code. Since Rascal integrates many aspects of code generation and program analysis into one language it is a natural choice for a project of this nature. Rascal also provides a very clean way to specify a concrete syntax and abstract syntax for a language, will automatically generate a parser for this language, and exports a simple interface for transforming a concrete syntax tree (CST) into an abstract syntax tree (AST). For example if one has specified a grammar called `Prog`, all that needs to be done to generate an AST for this grammar is:

```
public Prog parse(loc l) = parse(#Prog, l);
public Prog load(loc l) = implode(#Prog, parse(l));
```

In this example both the `parse(#Prog, l)` and `implode(#Prog, parse(l))` functions are built-in functions inside Rascal which will generate a CST and AST, respectively.

In addition to the features previously mentioned Rascal includes built-in features specifically for tree traversal and manipulation with the `visit` concept. This concept allows for automated traversal of a tree-like structure where each node is visited and operations can be performed on each node. These operations can include some form of side effect as well as modifying or replacing the current element, among others [1].

III. Solution

In order to accomplish the goals of this project we first had to design and implement two languages one containing syntactic sugar, and one core sugar-free language. Once these languages were constructed we then had to develop interpreters for each in order to verify that the results of the programs we were generating were indeed correct. After the interpreters for both languages were finished we developed a compiler that would translate programs in language F to programs in language S. The reason for this was to have some frame of reference as to what the final synthesised compiler should look like and how it should behave. Finally, once all of this was complete, we were able to begin development on the synthesiser that would automatically produce the code for compiling from language F to language S.

I. Languages F and S

As previously mentioned we developed two languages for this project. These two languages are extremely similar with only three key differences: language F allows for multiple bindings in a let expression, multiple parameters to a function and the addition of `else if` condition to the standard `if then else`. In addition to this, each language contains standard arithmetic operations (i.e. addition, subtraction, etc), standard comparators (less than, greater that, etc), function applications, and sequencing of expressions. A program in either language is made up of a number of global lets followed by an expression. Grammars for these languages can be found in Figure 1. Both of these languages were constructed in Rascal along with interpreters for each.

II. Compiler

There are two convenient aspects of Rascal that make the compiler between these two languages somewhat formulaic: function overloading and patterns as formal parameters. Function overloading works in the standard fashion of multiple functions which have the same name and only differ in the parameters. However, Rascal has the ability to use patterns as formal parameters which lets the programmer match different constructors to the same type as opposed to just different type. With these features the compiler, written in Rascal, and using Rascal's function overloading and pattern parameters is somewhat formulaic. For example consider the following code, used to compile addition in language F to addition in language S:

```
public lang::DFunc::AST::Exp comp(add(Exp e1, Exp e2)) = add(comp(e1),comp(e2));
```

Here we see a function, `comp()`, which takes, as a parameter, a pattern which matches the `add` node in language F and returns an `add` node in language S with each component of the `add` node compiled into a language S node. This a simple example since `add` nodes in language F are structured identically to `add` nodes in language S. A more complicated example takes function nodes, `func`, in language F and converts them to `func` nodes in language S. The code for this follows:

```
public lang::DFunc::AST::Exp comp(func([str f], Exp body)) = func(f,comp(body));
public lang::DFunc::AST::Exp comp(func([str f0, *str fn], Exp body)) =
           func(f0,comp(func(fn,body)));
```

⟨*Prog*⟩ ::= ⟨*Letg*⟩* ';;' ⟨*Exp*⟩

⟨*Letg*⟩ ::= 'let' '(' ⟨*Ident*⟩ ','* ')' '=' '(' ⟨*Exp*⟩ ','* ')' 'end'

⟨*Elif*⟩ ::= 'else if' ⟨*Exp*⟩ 'then' ⟨*Exp*⟩

⟨*Exp*⟩ ::= ⟨*Ident*⟩
| ⟨*Natural*⟩
| '(' ⟨*Exp*⟩ ')'
| 'let' '(' ⟨*Ident*⟩ ','* ')' '=' '(' ⟨*Exp*⟩ ','* ')' 'in' ⟨*Exp*⟩ 'end'
| 'if' ⟨*Exp*⟩ 'then' ⟨*Exp*⟩ ⟨*Elif*⟩* 'else' ⟨*Exp*⟩ 'end'
| 'fun' '(' ⟨*Ident*⟩ ','* ')' '->' ⟨*Exp*⟩
| ⟨*Exp*⟩ '*' ⟨*Exp*⟩
| ⟨*Exp*⟩ '/' ⟨*Exp*⟩
| ⟨*Exp*⟩ '%' ⟨*Exp*⟩
| ⟨*Exp*⟩ '+' ⟨*Exp*⟩
| ⟨*Exp*⟩ '-' ⟨*Exp*⟩
| ⟨*Exp*⟩ '=' ⟨*Exp*⟩
| ⟨*Exp*⟩ '>' ⟨*Exp*⟩
| ⟨*Exp*⟩ '⟨' <*Exp*⟩
| ⟨*Exp*⟩ '>=' ⟨*Exp*⟩
| ⟨*Exp*⟩ '⟨=' <*Exp*⟩
| ⟨*Ident*⟩ '(' ⟨*Exp*⟩ ','* ')'
| ⟨*Exp*⟩ ';' ⟨*Exp*⟩

⟨*Prog*⟩ ::= ⟨*Letg*⟩* ';;' ⟨*Exp*⟩

⟨*Letg*⟩ ::= 'let' '(' ⟨*Ident*⟩ ','* ')' '=' '(' ⟨*Exp*⟩ ','* ')' 'end'

⟨*Exp*⟩ ::= ⟨*Ident*⟩
| ⟨*Natural*⟩
| '(' ⟨*Exp*⟩ ')'
| 'let' ⟨*Ident*⟩ '=' ⟨*Exp*⟩ 'in' ⟨*Exp*⟩ 'end'
| 'if' ⟨*Exp*⟩ 'then' ⟨*Exp*⟩ 'else' ⟨*Exp*⟩ 'end'
| 'fun' ⟨*Ident*⟩ '->' ⟨*Exp*⟩
| ⟨*Exp*⟩ '*' ⟨*Exp*⟩
| ⟨*Exp*⟩ '/' ⟨*Exp*⟩
| ⟨*Exp*⟩ '%' ⟨*Exp*⟩
| ⟨*Exp*⟩ '+' ⟨*Exp*⟩
| ⟨*Exp*⟩ '-' ⟨*Exp*⟩
| ⟨*Exp*⟩ '=' ⟨*Exp*⟩
| ⟨*Exp*⟩ '>' ⟨*Exp*⟩
| ⟨*Exp*⟩ '⟨' <*Exp*⟩
| ⟨*Exp*⟩ '>=' ⟨*Exp*⟩
| ⟨*Exp*⟩ '⟨=' <*Exp*⟩
| ⟨*Ident*⟩ '(' ⟨*Exp*⟩ ','* ')'
| ⟨*Exp*⟩ ';' ⟨*Exp*⟩

Figure 1: *Grammars for languages F and S.*

Once again we leverage the pattern matching capabilities in Rascal to not only match the function call on a `func` node but also match the structure of that node. The function in the first line will only be executed when called with a `func` node from language F that has one formal parameter. It will then return a `func` node from language S with the body compiled into a language S node. The next two lines show the recursive case which will be called when a `func` node from language F has multiple parameters. This case will create a language S `func` node with the first parameter from the language F `func` node and recursively call the `comp` function on a new language F `func` node which contains a list of the remaining parameters and the body of the original function. The cases for `let` and `else if` follow similar logic.

III. Synthesiser

It is the goal of the synthesiser to take input programs and generate the code for the compiler described in the previous section. A pictorial representation of how the synthesiser should work can be found in Figure 2. The synthesiser works in two stages: reading input and code generation. As input the synthesiser will take programs written in language F, convert it to its corresponding AST, and use the structure of the AST to synthesise code which transforms programs of that style to programs in language S. In order to accomplish this we use Rascal's `visit` concept to traverse the language F tree and perform certain operations based on the structure of that node. Building off the previous example of compiling `add` nodes from language F to `add` nodes of language S we can give the synthesiser the following program:

```
;;
x+x
```

From this program we will generate the following AST: `add(exp(var(x)),exp(var(x)))`. When the synthesiser analyses the `add` node, it will notice that this is a node which contains two parts `exp()` and `exp()` and is called `add`. This will be used to help generate the parameters to this node's `comp` function and the type of node to convert to in order to make this a valid node in language S. Since there are two parts to this node, it will take a node called `add` which is comprised of two parameters. Next, the synthesiser will examine each part of the `add` node and use the type to generate the types that make up the `add` parameter and the types that make up the return value. Since both of these are of type `exp` the synthesiser will insert `Exp e1` and `Exp e2` as the parts of the `add` node in the parameter. Since each sub part of a language F node needs to be compiled every `exp` node needs to be converted into a recursively call on the matching `Exp` parameter.

For more complicated examples we introduce expanding lists into recursive calls. This allows use to translate nodes of the form `func([param_list],exp(body))` to the corresponding compilation functions.

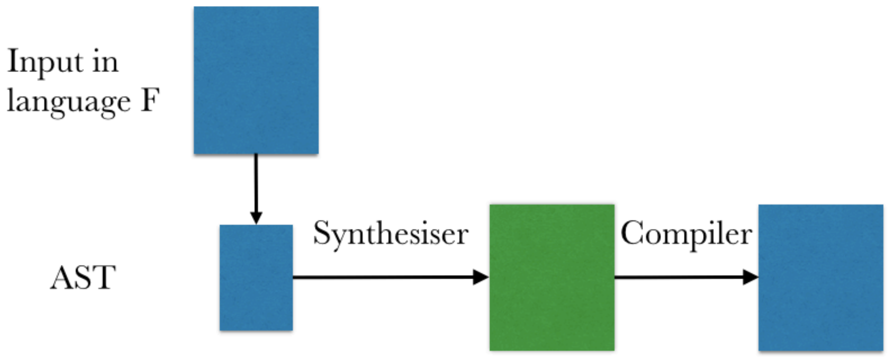

Figure 2: *Overview of synthesiser.*

IV. Evaluation

In order to evaluate the functionality of our synthesiser we produced a series of inputs in language F to be used by the synthesiser to generate the necessary parts of the compiler. Once we generated the compiler we wrote a number of programs in language F, compiled them using the synthesised compiler and evaluated each program using the previously developed interpreters. If all the results were the same then the synthesised compiler was determined to be correct. In all the test cases the results were indeed the same. One attractive aspect of this technique is the input programs needed to generate the compiler are very small.

V. Challenges

There were a number of challenges that had to be overcome, the first of this was the challenges presented by using Rascal. While there are a number of attractive features the language provides it is still in the alpha stages and documentation can be often lacking sufficient detail. This forced some decisions about the structure of the languages that weren't necessarily ideal, but were the only apparent ways to accomplish them in Rascal. This is, of course, in addition to developing two languages with the goal of compiling one language into the other.

The main challenge of the project was determining how to generalise the synthesiser enough to translate nodes of language F to nodes of language S without simply pattern matching each type of node and producing the corresponding `comp` function. As the project progressed into the final stages and some of the more complicated transformations were attempted, it became apparent the entire method of synthesis wasn't general enough. One possibility to generalise code the synthesiser generates even more is to provide two inputs for each transformation: one input in language F and one input in language S. The synthesiser would then analyse the input tree, from language F and synthesise code to translate nodes of that type to nodes of the type of language S. This is instead of trying to figure out all of the information from a single input in language F. This will also allow us to perform more complicated transformation, such as transformations from nodes of one type to nodes of a different type. A concrete example of this would be to translate `let` nodes in language F to simple `app` nodes in language S. In the current system this isn't possible because the synthesiser only has the information pertaining to the tree generated for the program in language F and doesn't have any information about what that program should be translated to. This is a fundamental flaw in the system.

VI. Future Work

There are number of very interesting directions this project could go from here. The first and foremost is to refine the languages to a more reasonable form. One idea would be to use a simple core lambda calculus as language S and slightly more complicated version for language F. The main focus of the future work will be to determine a viable method for using inputs from both languages to generate functions to compile programs from language F to language S. Being able to use the information from both types of programs will greatly increase the power of the synthesiser.

In addition to this the work can move on to synthesiser more advanced language features such as `switch` statements, `while` loops, etc. Lastly, this type of process could be used to try synthesising transformation on a 'real' language (i.e. a language actually used by programmers today such as Java, Javascript, etc).

References

[1] Rascal concepts: Visiting. `http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Concepts/Visiting/Visiting.html`.

[2] Rascal metaprogramming language. `www.rascal-mpl.org/`.

[3] 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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.