Experience with Higher Order Rewriting from the Compiler Teaching Trenches

Kristoffer H. Rose
Two Sigma[*] Labs and New York University

August 4, 2014

Abstract

We have now twice used the "HACS" compiler generator tool in an NYU graduate Compiler Construction class. HACS is based on a higher-order rewriting formalism, thus we have effectively been teaching students higher order rewriting techniques as the way to implement compilers. In this paper we report on how HACS matches specific rewriting notions to the techniques used by compiler writers, and where the main difficulties have been encountered in teaching these.

1 Introduction

NYU (New York University) offers a graduate so-called "Capstone" class in *Compiler Construction* [7]. The course is based on approximately a third of the material in the "dragon book" [1], and includes a compiler programming project. In class we focus on the formal techniques used to structure a compiler, as the dragon book goes to great length to use formal notions to describe and assist in the development of the various compiler components:

Regular Expressions are used to formalize the *lexical analysis* of (the text of) programs. The implementation of regular expressions by translation to deterministic finite automata (DFAs) is detailed.

Context Free Grammars are used to formalize the *syntax analysis* of programming languages, including how a program is converted into a parse tree. Algorithms for syntax analysis based on translating grammars into data structures that can control either top-down (LL) or bottom-up (LALR) parsing are explained.

Attribute Grammars (in variations called *Syntax-Directed Definitions* (SDDs) and – *Translations*) are used to formalize most aspects of *program analysis* and *program translation* (such as to intermediate code). General considerations on how one might "evaluate" an attribute grammar are considered, with remarks that one should really stick with specific

[*]

restricted categories of attribute grammars that can be evaluated with a simple left-to-right strategy ("L-attributed" for top-down and "S-attributed" for bottom-up parsing).

Graph Algorithms are finally presented in somewhat informal form as the technique of choice for several optimization problems.

In the class we present the view that the dragon book essentially defines the current practice of compiler writing, and we notice that while it includes systematic methods for lexical and syntax analysis, backed by tools that automate the process, no such methods are in common use for the remaining formal notions. Our goal is to retain and teach the original material in the dragon book yet offer tool support for the entire compiler program development experience to enforce the deep use of the used formalisms paired with an operational understanding of the involved algorithms.

Our answer to the issue raised above is the tool HACS, which stands for *Higher-order Attribute Contraction Schemes*. HACS allows the students to specify a complete compiler as a single source file, which combines the following formal notations:

- Lexical analysis based on usual *regular expressions* for tokens and white space.
- Syntax analysis based on a usual *context free grammar* with special higher-order features supporting creation of lexical scopes (and the related symbol tables) in addition to (usual) operator precedence mechanics.
- Propagation rules for *synthetic attributes*.
- Rules for specifying *recursive translation schemes* that serve the dual role of supporting *rewriting* and propagation of *inherited attributes*.

HACS is built on top of CRSX, which was developed and is used for implementing production compilers at IBM [8–10]; a fact not wasted on the students, many of whom are professionals pursuing an M.Sc. degree.

In this paper we discuss some details of the HACS file format, and comment on issues encountered in teaching these. In Section 2 we explain this for lexical analysis, and in Section 3 for syntactic analysis. Section 4 then discusses how semantic analysis is introduced, which covers explaining most of the HACS concepts. Throughout, we give examples from *first.hx* of the HACS distribution, which is a HACS version of [1, Chapter 2, Figure 7].

2 Lexical Analysis

Lexical analysis is the process of splitting the input text into tokens. HACS uses a fairly standard variation of *regular expressions* for this.

2.1 Example. Here are the simple regular expressions used to define the spacing conventions and terminal symbols (tokens) for the compiler of our small expression language.

```
space [ \t\n] ;

token Int   | ⟨Digit⟩+ ;
token Float | ⟨Int⟩ [.] ⟨Int⟩ ;
token Id    | ⟨Lower⟩+ ([_]? ⟨Int⟩)* ; // for symbols

token fragment Digit | [0−9] ;
token fragment Lower | [a−z] ;
```

The notation here is rather standard, and the only curious aspect is the use of the special Unicode angle brackets ⟨...⟩ to mark *inclusion* of other defined tokens and fragments. In addition, spaces are not significant in regular expressions, as is the case for the formal notation used in the dragon book. The lexical declarations also introduce the general shape of HACS units: a keyword is followed by the defined name and then several alternatives, all introduced by `|` (a vertical bar, like the way conditional cases are done in Haskell [5]), finally followed by a `;` (semi-colon). The `Id` token is equipped with the special tail "`([_]? ⟨Int⟩)*`," which permits the use of `Id` as a symbol token later on; this HACS-particular convention of course has to be explained. As is common with parser generators, otherwise unique tokens (such as keywords) can be declared "inline" in the syntax productions where they are used, as we shall see below.

3 Syntax Analysis

Once we have tokens, we can use HACS to program a complete syntax analysis with a grammar that specifies how the input text is decomposed to a *parse tree* according to a *concrete syntax* and how the desired *abstract syntax tree* (AST) is constructed from that.

With it's roots in term rewriting and algebraic specification, HACS focuses on the abstract syntax tree, with explicit declarations of the algebraic *sorts* that correspond to each kind of abstract syntax tree node. This means that HACS works with the paradigm that *all* concrete syntax issues are handled by "desugaring" or "disambiguation." For small and medium sized grammars this works well (we come back to the limitations below).

3.1 Example. Our small example source language merely has blocks, statements, and a few forms of expression. Here is a HACS grammar for this:

```
sort Name | symbol ⟦⟨Id⟩ ⟧;

sort Exp  | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
          | ⟦ ⟨Exp@2⟩ * ⟨Exp@3⟩ ⟧@2
          | ⟦ ⟨Int⟩ ⟧@3
          | ⟦ ⟨Float⟩ ⟧@3
          | ⟦ ⟨Name⟩ ⟧@3
          | sugar ⟦( ⟨Exp#1⟩ ) ⟧@3 →Exp#1 ;

sort Stat | ⟦ ⟨[x:Name]⟩ := ⟨Exp⟩ ; ⟨Stat[x:Exp]⟩ ⟧
          | ⟦ { ⟨Stat⟩ } ⟨Stat⟩ ⟧
          | ⟦ ⟧ ;
```

We explain the details of the notation below.

The grammar is explained as capturing several elements from the dragon book:

Literal syntax is indicated by the double "syntax brackets," ⟦...⟧. Text inside ⟦...⟧ consists of three things only: space, literal character "words," and references to nonterminals and predefined tokens inside (nested) ⟨...⟩. The literal syntax is taught as a variant of quasi-quoting or macro notation, common in many programming languages.

Symbol management is specified with the `symbol` annotation used in the `Name` sort declaration. This declares that the `Name` sort is special in that the system should rename

Name AST nodes to ensure that they are unique up to renaming, just as would be done in usual symbol table management.

Syntactic sugar is represented by the sugar part of the Exp sort declaration, which states that the parser should accept an Exp in parenthesis, identified as #1, and replace it with just that same Exp, indicated by the → Exp#1 part. This is explained as part of the mechanics between concrete and abstract syntax.

Precedence rules are represented by the @-annotations, which assign precedence to each operator. This is taught in the obvious way by explaining that the same Exp language would be recognized by the following concrete HACS specification (where we have also made the sugar concrete):

$$
\begin{array}{l}
\textbf{sort}\ \mathit{Exp}\ \ \mid [\![\ \langle \mathit{Exp1} \rangle\ ]\!]; \\
\textbf{sort}\ \mathit{Exp1} \mid [\![\ \langle \mathit{Exp1} \rangle + \langle \mathit{Exp2} \rangle\ ]\!] \mid [\![\langle \mathit{Exp2} \rangle\ ]\!]; \\
\textbf{sort}\ \mathit{Exp2} \mid [\![\ \langle \mathit{Exp2} \rangle * \langle \mathit{Exp3} \rangle\ ]\!] \mid [\![\ \langle \mathit{Exp3} \rangle\ ]\!]; \\
\textbf{sort}\ \mathit{Exp3} \mid [\![\ \langle \mathit{Int} \rangle\ ]\!] \mid [\![\ \langle \mathit{Float} \rangle\ ]\!] \mid [\![\ \langle \mathit{Name} \rangle\ ]\!] \mid [\![\ (\ \langle \mathit{Exp} \rangle\ )\ ]\!]\ ;
\end{array}
$$

One can then explain the abstract syntax tree as the conversion of the concrete syntax tree obtained with this explicitly layered grammar to the structure described by the HACS grammar in Example 3.1 with the sugar and @-annotations removed.

Lexical scoping and the associated renaming rules are specified as part of the syntax in HACS—this of course is part of what makes HACS a *higher order* formalism.

In our example, the first alternative of the Stat sort declaration defines a static scope with the use of [x:...]. The ⟨[x:Name]⟩ reference is the same as ⟨Name⟩ where we instead of inserting the token into the syntax tree just load it as a bound variable that we shall refer to as x. The ⟨Stat[x:Exp]⟩ reference is then the same as ⟨Stat⟩ except we also require that all the occurrences of the variable refererred to as x are considered to be instances of that bound variable. The declaration is allowed because the additional conditions are satisfied.

Of these, the lexical scoping, which is a kind of higher-order abstract syntax (HOAS), is of course the hardest to explain. It is explained as a mechanism to implement symbol tables with lexical and global scoping as well as allowing much easier access to scoped information in the later passes of the compiler. This is justified by showing how the students can experiment cleanly with the HOAS aspect without being distracted by all the other functions that the symbol table has, and the claims that the up front complexity pays back many times over, when the body of the compiler is to be written, is justified by proceeding right to such an example, described in the following section.

4 Semantic Analysis

In terms of higher order rewriting, the above sections provide algebraic sorts, defined by the syntax productions—in terms of the dragon book, and thus our students, a mechanism for generating abstract syntax trees (ASTs).

The next stage is to implement *semantic* analyses: rules for enriching the AST with additional information that facilitates the subsequent transformation to an intermediate representation (IR). We use type analysis as an example. Because type analysis uses the symbol table in a non-trivial way, the dragon book delays type analysis until later, however, in the class we use it as an early example of an analysis specified by a syntax-directed definition (SDD).

PRODUCTION	SEMANTIC RULES	
$S \rightarrow \textbf{name} := E_1; S_2$	$E_1.e = S.e; S_2.e = \text{Extend}(S.e, \textbf{name}.sym, E_1.t)$	(S1)
$\mid \{ S_1 \} S_2$	$S_1.e = S.e; S_2.e = S.e$	(S2)
$\mid \epsilon$		(S3)
$E \rightarrow E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	(E1)
$\mid E_1 * E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	(E2)
$\mid \textbf{int}$	$E.t = \text{Int}$	(E3)
$\mid \textbf{float}$	$E.t = \text{Float}$	(E4)
$\mid \textbf{name}$	$E.t = \text{if Defined}(E.e, \textbf{name}.sym) \text{ then Lookup}(E.e, \textbf{name}.sym)$ else TypeError	(E5)

Figure 1: SDD for type checking.

4.1 Example. An SDD for a simple type analysis can be implemented with two attributes (and using the usual convention that the SDD uses E and S where the HACS grammar has `Exp` and `Stat`):

- The inherited *environment* attribute e, which is a map from variables to types on both the statement and expression non-terminals S and E.
- The synthesized *type* attribute t on expressions E, which contains the type of the expression.

With those, the SDD can be expressed as shown in Figure 1, where we use the helpers Extend, Defined, and Lookup to build an extended environment with an additional type declaration, check for existence, and look one up, respectively, and Unif to find the type of an arithmetic operation with operands of two specific types.

The SDD in Example 4.1 is explained as a slightly more declarative version than the SDDs in the dragon book, where SDDs generally operate on the symbol table using side effects. It is not hard to justify to students why it is worthwhile to strive for *immutable* symbol tables and that this helps relaxing the evaluation order used to evaluate the attributes.

4.2 Example. The full HACS code of the type checker is shown in Figure 2.

The HACS code for the SDD of the example reveals most of the remaining HACS properties:

Algebraic sorts are available: the example defines the sort `Type` which contains our possible types as well as a so-called *scheme* which has associated signature as well as...

Rewrite rules can be given for schemes, here for example `Unif` in lines 6 through 9. A scheme can have a `default` rule used as a fall-back. *Meta-variables* in rewrite rules are denoted by a starting `#`.

Synthesized attributes are declared with an `attribute`↑ declaration followed by the attribute name and sort of the attribute value to propagate in `()`s. Line 11 is an example of this, declaring the attribute `t` of sort `Type`, and line 12 then declares that `t` is available on AST nodes of `Exp` sort.

Synthesis propagation rules are for synthesized attributes that depend exclusively on other synthesized attributes ("S-attributed") follow a specific scheme, illustrated by lines 14 through 17. Note that with HACS this can be the case for only *some* of the productions, as is the case here: the SDD productions (E1–E4) declare the `t` attribute in a way that only depends on subterm synthesized attributes. The pattern of such S-attributed propagation rules is as follows, illustrated by (E1). It appears like this in the SDD:

$E \to E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \mathrm{Unif}(E_1.t, E_2.t)$	(E1)

Here is how we teach converting the local S-attributed propagation of `t` to HACS form.

1. The first thing to do is copy the syntax production but omit any `@`-markings and add a unique `#n` disambiguation mark to each production. Since (E1) is based on the alternative

   sort *Exp* | ⟦ ⟨*Exp@1*⟩ + ⟨*Exp@2*⟩ ⟧@*1*

   we start with

   ⟦ ⟨*Exp*#1⟩ + ⟨*Exp*#2⟩ ⟧

   where we have used the subscripts from (E1) as `#`-disambiguation marks.

2. Next add in *synthesis patterns* for the attributes we are reading. Each attribute reference like $E_1.t$ becomes a pattern like ⟨`Exp#1 ↑t(#t1)`⟩, where the meta-variables like `#t1` should each be unique. For our example, this gives us

   ⟦ ⟨*Exp*#1 ↑*t*(*#t1*)⟩ + ⟨*Exp*#2 ↑*t*(*#t2*)⟩ ⟧

   which sets up `#t1` and `#t2` as synonyms for $E_1.t$ and $E_2.t$, respectively.

3. Finally, add in the actual synthesized attribute, using the same kind of pattern at the *end* of the rule (and add a `;`), and we get

   ⟦ ⟨*Exp*#1 ↑*t*(*#t1*)⟩ + ⟨*Exp*#2 ↑*t*(*#t2*)⟩ ⟧↑*t*(*Unif*(*#t1*,*#t2*)) ;

Synthesis rules are not hard for the students to get right. We make sure that the invariant is clear: the purpose of synthesis rules is to make it possible to match on a synthesized attribute for every subexpression of the appropriate sort. We also make sure that at this point the students take note of an rules that were *not* S-attributed and are thus still left out.

Inherited attributes are specified with an `attribute↓` declaration. This is similar to synthesized attributes except we also support *maps* such as symbol tables (but the keys do not have to be symbols). Value attributes are specified as for synthesized with an attribute name followed by the sort in `()`s, and map attributes are specified with the name followed by a `{`, the key sort, a `:`, the value sort, and a `}`. The e attribute of the SDD is declared in HACS in line 19 in this way as a map from `Name` keys to `Type` values, thus capturing the type of variables.

Recursive schemes are the final tool available. For our SDD we have to use recursive schemes for any semantic rule that involves any inherited attributes. The basic rules that we teach are these:

1. Every inherited attribute needs to be *associated* to a recursive scheme, which will then be responsible for propagating that inherited attribute.

   In general a scheme is declared with a `scheme` declaration, like the one in line 21, which otherwise looks like a syntax production alternative, and indeed *defines new syntax*. This has to be carefully explained to students: that we effectively extend the syntax of the user language—after line 21, the expression `TA 2` is a legal expression in the language, however, it is a fake expression that is used internally by the compiler. (This also has the advantage that one can more easily test internal schemes of the compiler by providing input "programs" that contain the schemes.)

   Line 21 also contains the association of the inherited `e` attribute to the `TA Exp` scheme: the ↓`e` suffix. This implies that *every application of the TA scheme will always propagate the inherited e attribute from its context*.

   Line 31 achieves the similar effect to the `TA` scheme for `Stat` nodes (the name can be the same as there is no syntactic overlap between `Stat` and `Exp`, a fact the students can understand from the parser material already treated in depth at this point). Only notice that to avoid ambiguity, the `TA` scheme for the `Stat` sort includes `{}`s as part of the syntax.

2. *Semantic rules with simple inheritance* then get translated to straight recursive traversal rules. An example of this is (E1), where the environment e is propagated straight to the two components E_1 and E_2. Line 28 captures this; indeed all of (E1–E4) satisfy this requirement, (E3–E4) trivially so, as captured by lines 26–29. Conversely for

   We here have to explain carefully how we exploit the syntax mechanism, so for example in line 28 the left of the $\rightarrow$ is a HACS pattern representation defining the application of `TA` to the expression $\mathsf{E}_1 + \mathsf{E}_2$, whereas the right of the $\rightarrow$ represents the `+`-`Exp` where the two subexpressions are applications of `TA` to the sub-`Exp` that correspond to E_1 and E_2, respectively (including the use of the syntactic sugar `(...)`). The recursive scheme fully represents the propagation of the inherited e attribye in (E1): the application of `TA` on the left side of $\rightarrow$ captures an e corresponding to $\mathsf{E}.e$ in (E1), and the two recursive applications of `TA` on the right side of $\rightarrow$ conversely represent passing "the e" into the corresponding subexpression, corresponding to the two attribute definitions $\mathsf{E}_1.e = \mathsf{E}.; \mathsf{E}_2.e = \mathsf{E}.e$.

   Here we also explain that it is posible to "expand" patterns such as the ones used in Line 28, which could have been written with nested references and literal syntax like this:

   $$\llbracket\ TA\ \langle Exp\ \llbracket\langle Exp\#1\rangle\ +\ \langle Exp\#2\rangle\rrbracket\rangle\ \rrbracket \\ \rightarrow \llbracket\ \langle Exp\ \llbracket TA\ \langle Exp\#1\rangle\rrbracket\rangle\ +\ \langle Exp\ \llbracket TA\ \langle Exp\#2\rangle\rrbracket\rangle\ \rrbracket;$$

   avoiding any use of the `()` sugar. It is sometimes a help for students to go through this a few times when they have issues in the code.

   Line 40 and 41 correspond in a similar simple way to the simple inheritance of (S2–S3).

3. *Semantic rules with synthetic depending on one inherited* are handled next. They are rules that synthesize some attributes depending only on other synthetic attributes (of subterms) and, crucially, depend on *one* inherited attribute. An example in the example SDD is (E5), which defines the t synthesized attribute in terms of the inherited e. This is represented by the rules in lines 23–24, which show how we

define the `TA` scheme (since we are using `e`) for the `Name` case: we have to use a parsed `Name`, `id`, and then we have two subcases, corresponding to the two cases of the Defined predicate from the SDD:

- the first case in line 23 captures that `e` has mapping from ⟦`id`⟧ (that is `id` parsed as a `Name`, the key sort) to some type, which is captured by `#t`. The `TA` scheme then takes care of the synthesis of `t` *and* the propagation of `e` (which is trivial in this case);
- the second case in Line 24 captures when `e` does not have a mapping, which should result in an error.

When this situation arises, which is common, then we get a *scheme dependency*: synthesis of the `t` attribute *depends on* having finished the `TA` scheme for the appropriate subterm.

4. Rules that *extend the map attribute* require a bit more notation. In our example, this is just (S1), which extends the e attribute that is passed down to S_2 with one more symbol to type mapping. Specifically, we have to explain how to encode what we have written as $\mathrm{Extend}(N_1.a, k, v)$ in the SDD notation, assuming the a inherited attribute is associated with a scheme, say the `A` scheme for the `S` sort. This is written as the special reference

   ⟨`S` ⟦`A` ⟨`N#1`⟩⟧ ↓`a{k:v}`⟩.

   We dissect this with the students:

   - The outermost ⟨`S`...⟩ just establishes that this is an `S`-unit in the context (which we assume to be parsed syntax).
   - The actual unit is computed by the `A` scheme on N_1, which we refer to by convention as ⟨`N#1`⟩, so the whole invokation is ⟦`A` ⟨`N#1`⟩⟧. If we were not using a changed `e` that would be all, but we are modifying the `e` attribute sent in, by adding a map, which is indicated by adding ↓`a{k:v}` for whatever `k` and `v` are.

   However, (S1) has one more special feature, which is the next point invesitigated.

5. Rules that *have left-to-right attribute dependencies* require that helper schemes are introduced. This is similar to the way one reasons about how to execute operational semantic inference systems "from left to right:" the intermediate state where only part of the attributes (premises for inference system) are instantiated needs to be captured.

   For rule (S1) this is present in that the extension of the inherited $S_2.e$ depends on the synthesized $E_1.t$, which in HACS depends on having computed `TA` on the `Exp` corresponding to E_1 because `t` depends on `TA`.

   This justifies the final block, lines 33–38, which consists of a "splitting" rule, line 33, that does nothing but invoke `TA` on the expression *as well* as a new helper scheme, `TA2`, invoked on the entire statement *with* the nested `TA` inside. `TA2` is then declared in line 35 (in the same sort and also propagating `e`). Line 36 has the `TA2` pattern, which now can safely require a synthesized `t` attribute on the embedded expression, captured by `#t1` in the pattern corresponding to $E_1.t$ in (S1). Line 37 has the result of `TA2` which is a statement where we recurse the `TA` scheme into the S_2 substatement with an extended environment, written as discussed above.

Throughout this some of the details of the HACS syntax matters, for example that the last `sort` declaration is remembered as "the current sort," which is what permits rules and some declarations to "know" what sort they are applied to.

We also have to communicate some of the general properties of rewriting: that there is no predefined strategy, so rules have to be explicitly mutually exclusive (with `default` rule as the exception).

Finally, it has been very helpful to work in a strongly (many-)sorted algebraic system, as it is often possible when students have a problem to break their rules and declarations down to check consistency.

5 Conclusion and Future Work

We have reported some of the details of what it has been like to use the HACS (Higher-order Attribute Contraction Schemes) formalism as the implementation language for the compiler project of our Compiler Construction class at NYU. The full class goes on to explain *code generation* using very similar techniques but rewriting the AST to a code AST instead of deriving a "code" attribute.

In conclusion it appears to definitely be possible to teach traditional compiler construction skills along with a very formal methodology for actually programming the compiler. In a sense it turns the class on it's head compared to similar classes: very often the formalism is the "exam question stuff" and the project programming is all low level hacking—in our case we often found that we had to weight the rather low level algorithmic stuff (automata, common subexpression elimination) in the homework and exam, since the formal methods were covered in the "programming" project.

Related Work. Many have observed that compiler formal systems are essentially algebraic or rewrite systems, and tools have been developed that offer formal compiler development, from VDM [2] through ASF+SDF [4]. However, every compiler teaching effort that we know of that uses these methods has effectively turned into a class *on* the tool rather than a class that just happens to use it (one possible exception to this may be classes in Amsterdam using the now very mature Rascal [6] system). For a great analysis of the issues with algebraic formal methods for coding systems such as compilers, see van den Bos *etal.* [3], however, a comparison is out of scope for this paper.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, , and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Inc, 2006.

[2] D. Bjørner and C.B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[3] J. van den Bos, M. A. Hills, P. Klint, T. van der Storm, and J. J. Vinju. Rascal: From algebraic specification to meta-programming. In V. Rusu and F. Durán, editors, *AMMSE 2011—Second International Workshop on Algebraic Methods in Model-based Software Engineering*, volume 56 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–32, 2011.

[4] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[5] Simon Marlow (editor). Haskell 2010 language report, 2010. URL: `https://www.haskell.org/onlinereport/haskell2010/`.

[6] KlintVinjuvanderStorm:scam2009. RASCAL: A domain specific language for source code analysis and manipulation. In *International Workshop on Source Code Analysis and Manipulation*, pages 168–177. IEEE Computer Society, 2009.

[7] Kristoffer Rose. Compiler construction. NYU Capstone Class, 2014. URL: `http://cs.nyu.edu/courses/spring14/CSCI-GA.2130-001/`.

[8] Kristoffer H. Rose. CRSX – an open source platform for experimenting with higher order rewriting. Presented in absentia at HOR 2007—`http://www.krisrose.net/papers`, June 2007.

[9] Kristoffer H. Rose. CRSX – combinatory reduction systems with extensions. In Manfred Schmidt-Schauß, editor, *RTA '11—22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Novi Sad, Serbia, June 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.RTA.2011.81`.

[10] Kristoffer H. Rose. Higher-order rewriting for executable compiler specifications. In Eduardo Bonelli, editor, *HOR '10—Proceedings of the 5th International Workshop on Higher-Order Rewriting,* Edinburgh, Scotland, July 14, 2010, volume 49 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–45. Open Publishing Association, 2011. `doi:10.4204/EPTCS.49.3`.

```
// TYPE ANALYSIS

sort Type | Int | Float | TypeError
          | scheme Unif(Type,Type) ;

Unif(Int, Int) → Int;
Unif(#t1, Float) →Float;
Unif(Float, #t2) →Float;
default Unif(#1,#2) →TypeError;

attribute↑t(Type); // synthesized expression type
sort Exp | ↑t;

⟦ (⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩) ⟧↑t(Unif(#t1,#t2));
⟦ (⟨Exp#1 ↑t(#t1)⟩ * ⟨Exp#2 ↑t(#t2)⟩) ⟧↑t(Unif(#t1,#t2));
⟦ ⟨INT#⟩ ⟧↑t(Int);
⟦ ⟨FLOAT#⟩ ⟧↑t(Float);

attribute↓e{Name:Type}; // inherited type environment

sort Exp | scheme ⟦TA ⟨Exp⟩ ⟧↓e ; // propagates e over Exp

⟦ TA id ⟧ ↓e{⟦id⟧ : #t} →⟦ id ⟧ ↑t(#t);
⟦ TA id ⟧ ↓e{¬⟦id⟧} →error⟦Undefined identifier ⟨id⟩⟧;

⟦ TA ⟨INT#⟩ ⟧→⟦ ⟨INT#⟩ ⟧;
⟦ TA ⟨FLOAT#⟩ ⟧→⟦⟨FLOAT#⟩ ⟧;
⟦ TA (⟨Exp#1⟩ + ⟨Exp#2⟩) ⟧→⟦(TA ⟨Exp#1⟩) + (TA ⟨Exp#2⟩) ⟧;
⟦ TA (⟨Exp#1⟩ * ⟨Exp#2⟩) ⟧→⟦(TA ⟨Exp#1⟩) * (TA ⟨Exp#2⟩) ⟧;

sort Stat | scheme ⟦TA { ⟨Stat⟩ } ⟧↓e ;  // propagates e over Stat

⟦ TA { id := ⟨Exp#1⟩; ⟨Stat#2[⟦id⟧]⟩ } ⟧→⟦TA2 { id := TA ⟨Exp#1⟩; ⟨Stat#2[⟦id⟧]⟩ } ⟧;
{
  | scheme ⟦TA2 { ⟨Stat⟩ } ⟧↓e;
  ⟦ TA2 { id := ⟨Exp#1 ↑t(#t1)⟩; ⟨Stat#2[⟦id⟧]⟩ } ⟧→
    ⟦ id := ⟨Exp#1⟩; ⟨Stat ⟦TA {⟨Stat#2[⟦id⟧]⟩}⟧ ↓e{⟦id⟧:#t1}⟩ ⟧;
}

⟦ TA { { ⟨Stat#1⟩ } ⟨Stat#2⟩ } ⟧→⟦{ TA { ⟨Stat#1⟩ } } TA { ⟨Stat#2⟩ } ⟧;

⟦ TA { } ⟧ → ⟦ { } ⟧;
```

Figure 2: HACS code for type analysis.