# Higher-order Attribute Contraction Schemes
## *User's Guide*[*]

Kristoffer H. Rose

September 18, 2013

**Abstract**

HACS is a language for programming compilers. This guide introduces the basic features of HACS. HACS is distributed as part of the CRSX system [2].

## Contents



[*]**DRAFT—Feedback appreciated!** CVS Id: guide.tex,v 3.5 2013/09/18 14:42:43 krisrose Exp .

# 1 Introduction

With HACS it is possible to create a fully functional compiler by writing a single source file.

## 1.1 Running the example

Get the dragon

intended as a fully formalized yet pedagogical platform for implementing systems for "semantics-based program transformations" such as compilers and optimizers.

## 1.2 A Taste

For a taste of the system, here is a small example file [2, samples/guide/fupz.hx].

```
1  // HACS example: "Functions Using Plus Zero"
2  module FUP {
3   sort E | ⟦0⟧ | ⟦ ( ⟨E⟩ + ⟨E⟩ ) ⟧ | ⟦ ⟨F⟩ ⟨E⟩ ⟧ ;
4   sort F | ⟦ [ ⟨[V]⟩ ↦ ⟨E[V]⟩ ] ⟧ ; //abstraction
5   token V | [a-z]+[0-9]* ;

6   sort E | scheme ⟦run ⟨E⟩⟧;
7   ⟦run 0⟧ → ⟦0⟧;
8   ⟦run (⟨E#⟩+0)⟧ → ⟦run ⟨E#⟩⟧;
9   ⟦run ([x↦⟨E#[x]⟩] ⟨E#2⟩⟧ → ⟦run ⟨E#[E#2]⟩⟧;
10 }
```

The example illustrates these points:

- Comments are possible.
- A specification is always a module using the ubiquitous {}s.
- The specification is structured into "sorts" linked to grammar productions.
- Full Unicode can be used in general, and in particular concrete syntax is in ⟦...⟧ and grammar recursion in ⟨...⟩.
- The ⟦⟨[V]⟩ → ⟨E[V]⟩⟧ is special syntax for declaring higher order syntax, which will be explained.[1]
- Tokens are declared with regular expressions.
- Transformations are called "schemes" and also have concrete syntax.
- Schemes are specified with *rules* of the form "pattern" → "result," also using syntax.
- The last declaration illustrates how a rule can match higher order syntax and even perform substitution of bound variables.

## 1.3 First Run

## 1.4 Overview

In the rest of this document we introduce the most important features of HACS specifications.

[1] The special syntax for abstractions is explained in Section 2.2.

**Syntax.** Used to specify the external syntax of all representations used in files and rules. Structured around *productions* (*aka. syntactic sorts*) with additional notation for *concrete syntax sugar* so a parser can be built to map from text into abstract syntax, and *syntax formatting instructions* so a pretty-printer can be built to map from each used abstract syntax production back to text. The syntax supported by HACS is peculiar in that it can construct uses *higher order data*, which means that it supports parsing that includes variable "binders" and "occurrences" natively. Section 2 below explains the basic syntax mechanism, and further details are given in the reference Appendix **??**.

**Translation Schemes.** Base rules for how terms of the (abstract) syntax are transformed between the used intermediate representations, possibly depending on attribute annotations resulting from analysis. Typical translation schemes in a compiler are "normalization" and "code generation." Section 3 explains this, again with support in the reference Appendix **??**.

**Attribution Rules.** Rules for how attributes are assigned, in various forms, including *attribute grammars*, *deterministic inference rules*, and *tables*. Attributes typically are a combination of abstract syntax and values of *semantic sorts*; an example attribution would be "type assignment," which assigns a type attribute to each subexpression. Section 4 explains this with support in the reference Appendix **??**.

**Optimization Equivalences.** Finally, an experimental feature is to indicate the preferred form of semantically equivalent expressions, which can then be used to modify the translation schemes to be optimizing in a safe way. This is documented by Section 4 and Appendix **??**.

# 2 Syntax

HACS works with *sorted* abstract syntax trees (AST). Specifically, HACS provides a special notation for specifying abstract and concrete syntax together with the used sorts corresponding to the abstract syntax productions (and thus the AST is what is called a *many-sorted algebraic term*). In this section we explain the common syntax of the combined sort and grammar specifications.

## 2.1 First Order Syntax

We start off with the basic (first order) notational conventions. (The list is rather long but readers familiar with *parser generation* will hopefully find most of this section unsurprising, and we proceed with an example right afterwards.)

**2.1.1 Notation.** HACS uses the following notations for specifying the syntax to use for terms.[2]

- HACS *production names* are capitalized, so we can for example use `Exp` for the production of expressions. The name of a production also serves as the name of its *sort*, *i.e.*, the semantic category that is used internally for abstract syntax trees with that root production. If particular instances of a sort need to be referenced later they can be *disambiguated* with an `#`$i$ suffix, *e.g.*, `Exp#2`, where $i$ is an optional number or other label.
- A sort is declared by `sort` declarations of the name optionally followed by a number of *abstract syntax production* alternatives, each starting with a `|`.

[2]The notations use special Unicode characters: see the grammar in the appendix for their codes and alternatives.

- Double square brackets $[\![ \ldots ]\!]$ are used for *concrete syntax* but can contain nested angle brackets $\langle \ldots \rangle$ with *production references* like $\langle$`Exp`$\rangle$ for an expression (as well as several other things that we will come to later). We for example write $[\![\langle$`Exp`$\rangle$+$\langle$`Exp`$\rangle]\!]$ to describe the form where two expressions are separated by a + sign.

- A trailing `@`$p$ for some precedence integer $p$ indicates that either the subexpression or the entire alternative (as appropriate) should be considered to have the indicated precedence, with higher numbers indicating higher precedence, *i.e.*, tighter association. (For details on the limitations of how the precedence and left recursion mechanisms are implemented, see Appendix A.2.)

- `sugar` $[\![ \ldots ]\!] \to \ldots$ alternatives specify equivalent forms for existing syntax: anything matching the left alternative will be interpreted the same as the right one (which must have been previously defined); references must be disambiguated.

- `token` declarations define production of a single *string token* specified by a regular expression; similarly `space` defines the regular expression for the special token that is permitted and ignored between other tokens. (The token and regular expression formats are quite extensive and detailed in Appendix A.1, including how one deals with lookahead, keywords, special modes, nested comments, *etc.*)

- HACS supports C/Java-style comments, either with `//` for single line comments or `/*...*/` for delimited comments (may be multi-line and nested).

With these we can handle much of traditional parser generation.

**2.1.2 Example.** Numeric expressions are often specified with an informal syntax description like the following. First a *abstract syntax* in formal (non-HACS) notation:

$$e ::= e + e \mid e - e \mid e \times e \mid \frac{e}{e} \mid fe \mid n \qquad \text{(Exp)}$$

$$f ::= \sin \mid \cos \mid \ln \mid \exp \qquad \text{(Fun)}$$

$n$ ranges over non-negative number literals (that we are not particularly interested in here except that they are expected to include $0$ and $1$). In addition, we suppose standard disambiguation of nested expressions by the conventions that

- parentheses can be used, making $(e)$ a valid expression;

- parenthesis have highest precedence, and function application has higher precedence than $\times$, which in turn has higher precedence than $+$ and $-$ (no precedence is given for fraction expressions as the notation takes care of that);

- a sequence of operators with the same precedence associates to the left; and

- in a $\pm$-sequence one can omit a leading $0$.

So $-1 + 2 - 3 \times \frac{4}{\ln 5}$ is the same expression as $((0 - 1) + 2) - (3 \times (\frac{4}{\ln 5}))$.

We can use a very similar grammar as a HACS specification, except we limit ourselves to the more usual textual / for fractions, and choose a specific precedence numbering.

```
1  sort Exp
2  | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
3  | ⟦ ⟨Exp@1⟩ - ⟨Exp@2⟩ ⟧@1
4  | ⟦ ⟨Exp@2⟩ * ⟨Exp@3⟩ ⟧@2
5  | ⟦ ⟨Exp@2⟩ / ⟨Exp@3⟩ ⟧@2
6  | ⟦ ⟨Fun⟩ ⟨Exp@4⟩ ⟧@3
7  | ⟦ ⟨Int⟩ ⟧@4

8  | sugar ⟦ ( ⟨Exp#⟩ ) ⟧@4 → ⟦ ⟨Exp#⟩ ⟧
9  | sugar ⟦ + ⟨Exp#@2⟩ ⟧@1 → ⟦ 0 + ⟨Exp#⟩ ⟧
10 | sugar ⟦ - ⟨Exp#@2⟩ ⟧@1 → ⟦ 0 - ⟨Exp#⟩ ⟧
11 ;

12 sort Fun
13 | ⟦sin⟧@2 | ⟦cos⟧@2 | ⟦ln⟧@2 | ⟦exp⟧@2 ;

14 token Int | [0-9]+ ;
```

The effect of the declaration is that we achieve the capability to parse expressions with parentheses and all the variations of the original syntax yet the resulting abstract syntax behaves as if it had been specified with just

```
sort Exp|⟦⟨Exp⟩+⟨Exp⟩⟧|⟦⟨Exp⟩-⟨Exp⟩⟧
|⟦⟨Exp⟩*⟨Exp⟩⟧|⟦⟨Exp⟩/⟨Exp⟩⟧|⟦⟨Fun⟩⟨Exp⟩⟧|⟦⟨Int⟩⟧;
```

which are then all the `Exp` patterns that we need to be concerned with instantiating for compilation schemes and attributions below.

Our example expression from before can be written as `-1+2-3*(4/ln 5)` and will be parsed by HACS into an abstract syntax tree like... *To Be Done*... **TBD**

Finally, we remark that we have given the predefined functions precedence `@1` so we can add function expressions later.

## 2.2 Higher Order Syntax

We also need a way to express functional expressions, which require the higher order aspects of HACS, and is our first extension of the ⟨⟩-notation from above. This is used for managing *binding* (also known as *scope management tables*), which comes from the traditions of *λ-calculus* and *higher-order abstract syntax*.

**2.2.1 Notation.** HACS has special support for parsing bindings and bound occurrences of variables. These *will* be part of the abstract syntax in the form of *higher order data*, *i.e.*, data with embedded bound variables.

- The notation ⟨`[Var]`⟩ means that a `Var` token is parsed and introduced as a *binder* for some scope (to be specified separately);
- ⟨`Exp[Var]`⟩ specifies that the subexpression generated by ⟨`Exp`⟩ is in fact the *scope* of the binder variable named by the `Var` token (which must have exactly one binding specified in the alternative;[3] use the `Sort#`$i$ form of token sorts if disambiguation is necessary); and *To Be Done... Consider using `[Var]Exp` or `Var.Exp`.* **TBD**

[3] Currently HACS imposes the limitation that the binding specification must be to the left of the scope.

- Just using ⟨Var⟩ (as before) for a token that has binders means that the Var token will be identified with any bound tokens with the same text, if such exist for which it is in scope, and will then correspond to an *occurrence* of the variable.

**2.2.2 Example.** We extend the numeric functional expressions of Example 2.1.2 with a symbolic notation for functions:

$$e ::= \ldots \mid x \qquad \text{(Exp-2)}$$
$$f ::= \ldots \mid [x \mapsto e] \qquad \text{(Fun-2)}$$

where $x$ should be a *variable* (lower case letter), which is then *bound* inside the following $e$ expression in the usual way. We add this to the HACS code with the additional declarations

```
1 sort Exp | ⟦ ⟨Var⟩ ⟧@4 ;
2 sort Fun | ⟦ [ ⟨[Var#1]⟩ ↦ ⟨Exp[Var#1]⟩ ] ⟧ ;
3 token Var | [a-z] ;
```

Specifically, ⟦[⟨[Var#1]⟩↦⟨Exp[Var#1]⟩]⟧ should be read as follows:

1. A concrete [ character.
2. A Var token, which is introduced as a *binder*, indicated by the surrounding [] and disambiguated for later by adding #1.
3. A concrete ↦ mapping arrow (unicode code point \u21a6)
4. An Exp expression which is the *scope* for the Var binder from before, *i.e.*, wherein all occurrences of Var tokens identical to the one in the trailing []s (namely Var#1 from item 2) are considered *bound occurrences*.
5. A concrete ] character.

The example also illustrates that a sort can have many separate declarations that all contribute to the definition of the sort: the declarations here add to the ones from Example 2.1.2.

*To Be Done. . . Example with Kleene star/plus.* **TBD**

# 3 Translation Schemes

Once we have one or more abstract syntaxes to work with, we are ready for specifying *schemes* for how the source abstract syntax translates to the target abstract syntax.

## 3.1 Declaration

Schemes can be specified either using custom syntax or using the built-in term syntax. In its simplest form, a scheme is specified by simply declaring the syntax form of it with the keyword scheme, and then give a number of *rewrite rules* that define what the replacement should be for each variation of the scheme.

**3.1.1 Notation.** A *scheme* is just a syntactic form marked with the prefix scheme that is permitted to have rewrite rules that define its behaviour, typically what instances of the scheme should be transformed to.

**3.1.2 Example.** The *derivative* (in Lagrange notation) of a function $f$ is written $f'$. We can declare this as a scheme with

```
1  sort Fun | scheme ⟦⟨Fun@2⟩'⟧@2 ;
```

The precedence is to explicitly allow left recursion, so we can write `sin''`. (In this and several following examples in this section, we are essentially encoding a higher order rewrite system for symbolic differentiation due to Knuth [1, p.337] as a HACS specification over the numeric expressions from Example 2.1.2 and 2.2.2.)

Once we have a declared scheme then we need to populate it with transformation rules. This can be done in a number of ways.

## 3.2 Rewriting Rules

The principal way of declaring a scheme is with rewrite rules.

**3.2.1 Notation.** A *rewrite rule* has the following structure:

- A basic rule consists of a *pattern* that is a fully disambiguated instance of a scheme (to the left), then an arrow ($\rightarrow$), and finally a *contraction* (to the right), which must be a term of the same sort and where all inserted references must refer to disambiguated references in the pattern. The intention is that the rules for a scheme fully explain how the defined form is eliminated.
- Rules sometimes have a *rule prefix* of the form *name*(*options*) : in front of the rule; we will show a few examples of this along the way.

Before we continue with our higher order example, here is a simple first order one.

**3.2.2 Example.** We will be computing with logic, so here are some notation and simplification schemes for booleans.

```
1  sort B
2  | ⟦ t ⟧@4 | ⟦ f ⟧@4 | sugar ⟦ (⟨B#⟩) ⟧@4→B#

3  | scheme ⟦ ⟨B@2⟩ ∨ ⟨B@1⟩ ⟧@1;
4  ⟦t∨⟨B⟩⟧ → ⟦t⟧;  ⟦f∨⟨B#⟩⟧ → B#;

5  | scheme ⟦ ⟨B@3⟩ ∧ ⟨B@2⟩ ⟧@2;
6  ⟦t∧⟨B#⟩⟧ → B#;  ⟦f∧⟨B⟩⟧ → ⟦f⟧;

7  | scheme ⟦ ¬ ⟨B@3⟩ ⟧@3;
8  ⟦¬t⟧ → ⟦f⟧;  ⟦¬f⟧ → ⟦t⟧;
```

The example combines all the first order features we have seen: data values (for true and false), syntactic sugar (parenthesis), and defined schemes with rewrite rules (boolean connectives).

**3.2.3 Example.** The *derivative* (in Lagrange notation) of a function $f$ is written $f'$. The following rewrite rules define the derivative as a scheme with rules for how each the function forms of Example 2.1.2 and 2.2.2 transform into another form that corresponds to the derivative. Each rule handles an instance of the scheme for each of the possible instantiation of the `Fun` subterm. We first declare that we will now (continue to) populate the `Fun` sort.

```
1  sort Fun ;
```

We then define the transform for each possible form of the scheme; it is important that each rule pattern is strictly consistent with the grammar rules: each left hand side is an instance of the defined scheme ⟦⟨Fun⟩'⟧ and each right hand side a value of sort Fun; in addition the rules fully cover all forms of the scheme.

```
1  ⟦sin'⟧ → ⟦cos⟧ ;
2  ⟦cos'⟧ → ⟦[a ↦ - sin a]⟧ ;
3  ⟦ln'⟧  → ⟦[z ↦ 1/z]⟧ ;
4  ⟦exp'⟧ → ⟦exp⟧ ;

5  ⟦[x ↦ ⟨Exp#[x]⟩]'⟧ → ⟦[y ↦ D(y)[z↦⟨Exp#[z]⟩]]⟧ ;
```

Notice how cos and ln have function expressions as their derivative, *e.g.*, for cos the derivative ⟦[z↦1/z]⟧ denotes the numeric function with a bound variable identified by the Var token z and an Exp subexpression which is the division 1/z. When a bound variable like z is introduced in this way, HACS ensures that it is suitably named to be different from all other variables in scope. The last rule, the derivative of a function expression [x↦⟨Exp[x]⟩], is defined in terms of a "dependent derivative" and will be explained with Example 3.2.4 below.

**3.2.4 Example.** In Euler notation the derivative of an expression with respect to a dependent variable is written $D_x f$. We can represent this in HACS as follows:

```
1  sort Exp | scheme ⟦ D ⟨Exp⟩ [⟨[Var#1]⟩↦⟨Exp[Var#1]⟩] ⟧ ;
```

where the D-construction has two arguments: the dependent variable (of sort Exp) from the context, in ()s, and the function expression. Now for the derivative of each expression form. First constants and variables.

```
2  ⟦ D⟨Exp#⟩[x↦⟨Int#⟩] ⟧ → ⟦0⟧ ;

3  Bound:           ⟦ D⟨Exp#⟩[x↦x] ⟧ → ⟦1⟧ ;
4  Indep(Free(y)): ⟦ D⟨Exp#⟩[x↦y] ⟧ → ⟦0⟧ ;
```

The last two rules have an optional rule prefix, used to identify the rule and, for the last, declare with an option that a free variable is intended in the rule. The patterns of the rules are almost identical, different only on the form of the scope of x; this is a common pattern. Specifically, there are two rules for variables: one matching an instance of the dependent variable and one for a variable that is different, which is ensured with our first example of an option, the Free(y) indicator, which restricts y to variables that are not bound in the pattern, thus in particular not x.

We continue with the usual binary operations.

```
5  ⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩+⟨Exp#2[x]⟩] ⟧ → ⟦D⟨Exp#⟩[y↦⟨Exp#1[y]⟩] + D⟨Exp#⟩[z↦⟨Exp#2[z]⟩]⟧ ;
6  ⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩-⟨Exp#2[x]⟩] ⟧ → ⟦D⟨Exp#⟩[y↦⟨Exp#1[y]⟩] - D⟨Exp#⟩[z↦⟨Exp#2[z]⟩]⟧ ;
```

These illustrate one way bound variables are managed by HACS rules. Specifically, the pattern (left hand side) fragment x↦⟨Exp#1[x]⟩+⟨Exp#2[x]⟩ specifies that the pattern will match scopes that bind some variable, which we shall refer to as x, and that the scope is a +-expression with two sub-Exp terms that can both contain the variable. In the contraction (right hand side) we then *split* the scope in two. The first scope that is created is y↦⟨Exp#1[y]⟩, which has the new binder y and a copy of what matched Exp#1 before, except all occurrences of x are replaced by y. (In some textbooks such a replacement would have been written {y/x}Exp#1.) This ensures that the scope

is split cleanly, with the two new scopes over `y` and `z` completely disjoint. Such "scope splitting" is immensely useful, and is the reason we separated the dependent variable from the context and the bound variable in the scopes. (In fact this is the simplest case of a *substitution* that the following rules will have deeper examples of.)

The product and division rules are slightly more involved.

```
7  ⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩ * ⟨Exp#2[x]⟩] ⟧
8   → ⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩] * ⟨Exp#2[#]⟩ + ⟨Exp#1[#]⟩ * D⟨Exp#⟩[x↦⟨Exp#2[x]⟩] ⟧ ;

9  ⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩ / ⟨Exp#2[x]⟩] ⟧
10  → ⟦ ( D⟨Exp#⟩[x↦⟨Exp#1[x]⟩] * ⟨Exp#2[#]⟩ - ⟨Exp#1[#]⟩ * D⟨Exp#⟩[x↦⟨Exp#2[x]⟩] )
11        / (⟨Exp#2[#]⟩ * ⟨Exp#2[#]⟩) ⟧ ;
```

In these rules the pattern (left hand side) matches several separate units, `Exp#`, `Exp#1[x]`, and `Exp#2[x]`, just like the sum and difference rules. But here the contraction (right hand side) combines them in a more complex way than merely creating new scopes. Specifically, we have instances of `Exp#1[#]` and `Exp#2[#]` in the contraction. These are *proper substitutions*: if the pattern matched `Exp#1[x]` then the contraction `Exp#1[#]` means "create copy of what matched `Exp#1` except substitute all occurrences of the bound variable `x` with a copy of what matched `(Exp)#`." (Again, a traditional formal way of writing such a substitution would be `{#/x}#1`.)

Finally, the chain rule.

```
12  ⟦ D(⟨Exp#⟩)[x↦⟨Fun#⟩⟨Exp#[x]⟩] ⟧ → ⟦ ⟨Fun#⟩' ⟨Exp#[#]⟩ * D(⟨Exp#⟩)[x↦⟨Exp#[x]⟩] ⟧ ;
```

Note how the contraction uses the `'` notation from Example 3.2.3 to differentiate the function. Furthermore, the chain rule illustrates one last point: *omission* of bound variables: the pattern contains `x` ↦ ⟨`#Fun`⟩ ⟨`Exp#[x]`⟩, where both `Fun#` and `Exp#[x]` are in the scope of `x` yet only `Exp#[x]` has a marker indicating to keep track of references to the `x` bound variable. This is significant, and indicates that the pattern only matches if `#Fun` *does not contain instances of* `x`. Formally, `#Fun` is said to be *weak* for `x`, and this has the advantage that in the contraction we can use `#Fun` safely outside of any scope, as the chain rule indeed needs.

We can now rewrite. . . *To Be Done*. . . **TBD**

# 4 Attributions

In addition to translation schemes that traverse and create abstract syntax trees, compilers also make use of analysis to enrich the abstract syntax tree with *attributes* that are used to assist the translation schemes.

## 4.1 Adding Attributes to Terms

Attributes come in two flavors: *inherited* attributes that are used to attach additional information to schemes, and *synthesized* attributes that represent properties of data. (If you are familiar with *attribute grammars* then everything in this section should be standard.)

**4.1.1 Notation.** *Attributes* are added to terms using the following notations:

- Simple *synthesized attributes* are added after a data term as ↑*name*(*value*), where the *name* is a constant identifier and the *value* can be any term.

- Simple *inherited attributes* are added similarly after a scheme term as ↓*name*(*value*).
- An *indexed synthesized attribute* is added after a data term as ↑*name*{*key*:*value*}, where the *key* is either a variable or a constant.
- An *indexed inherited attribute* is added to a scheme term as ↓*name*{*key*:*value*}.

## 4.2 Declaring Attributes

Note that whent using attributes inside ⟦⟧ brackets then they must be inside the unparsed fragments, *i.e.*, inside ⟨⟩s.

**4.2.1 Notation** (attribute sorts)**.** Atributes can be declared with special declarations like these after a usual `sort HostSort` declaration:

- `| ↑Name(ValueSort)` defines that `HostSort` has the simple synthesized attribute `Name` which has `ValueSort` values.
- `| ↓Name(ValueSort)` similarly for an simple inherited attribute.
- `| ↑Name{KeySort:ValueSort}` defines that `HostSort` has the indexed synthesized attribute `Name` which for each constant or variable of `KeySort` has a distinct `ValueSort` value.
- `| ↓Name{KeySort:ValueSort}` similarly for an indexed inherited attribute.

## 4.3 Computing Attributes

**4.3.1 Example.** Consider our expressions from Examples 2.1.2, 2.2.2, 3.1.2, 3.2.3, and 3.2.4; in addition consider the booleans of Example 3.2.2 loaded. One piece of information that may be of interest for the derivative is to analyze whether an expression is always defined, *i.e.*, defined for all possible values of all variables. We can represent this information with a HACS simple synthesized attribute.

```
1  sort Exp | ↑ TOTAL(B);
```

The declaration adds the synthesized `TOTAL` attribute to the `Exp#` sort. We then provide information to HACS on how the attribute is synthesized for all `Exp#` forms. For constants and most operations this is simple.

```
2  ⟦⟨Int#⟩⟧ ↑ TOTAL(⟦t⟧) ;
3  ⟦⟨Var#⟩⟧ ↑ TOTAL(⟦t⟧) ;

4  ⟦⟨Exp#1↑TOTAL(B#1)⟩ + ⟨Exp#2↑TOTAL(B#2)⟩⟧ ↑ TOTAL(⟦ ⟨B#1⟩ ∧ ⟨B#2⟩ ⟧) ;
5  ⟦⟨Exp#1↑TOTAL(B#1)⟩ - ⟨Exp#2↑TOTAL(B#2)⟩⟧ ↑ TOTAL(⟦ ⟨B#1⟩ ∧ ⟨B#2⟩ ⟧) ;
6  ⟦⟨Exp#1↑TOTAL(B#1)⟩ * ⟨Exp#2↑TOTAL(B#2)⟩⟧ ↑ TOTAL(⟦ ⟨B#1⟩ ∧ ⟨B#2⟩ ⟧) ;
```

For division this is slightly harder, in that division is only total when the divisor is known to never be zero. For this we invent an additional boolean synthesized attribute, `NZ` for "Never Zero," to be developed in Example **??**.

```
7  ⟦⟨Exp#1↑TOTAL(#1)⟩ / ⟨Exp#2↑TOTAL(#2)↑NZ(#nz⟩⟧ ↑ TOTAL(⟦⟨B#1⟩∧⟨B#2⟩∧⟨B#nz⟩⟧) ;
```

*To Be Done. . . Reverse disambiguation. . .* **TBD**

```
8  ⟦⟨#Fun⟩ ⟨Exp# ↑TOTAL(#)⟩⟧ ↑ TOTAL(#) ;
```

### 4.4 Inference Rules

Many published analyses use some logic inference system with rules for proving the existence of and typically also deriving the value of some desired property. HACS supports a special syntax for entering such systems... *To Be Done*... **TBD**

### 4.5 Tables

In some cases a table format is preferable; this is supported in HACS by... *To Be Done*... **TBD**

## 5 Data Equivalences

Translation schemes like our derivation example can sometimes use a litle help. One thing that can happen, for example, is that an expression translates to a constant, which creates situations where arithmetic simplification is possible. This is supported by HACS systems through *data equivalences*, which are simplifications of *non-scheme* terms, which will be folded into the defined schemes.

### 5.1 Unconstrained Equivalences

**5.1.1 Invariant.** Equivalences must be equivalent for all translation schemes...

**5.1.2 Example.** Arithmetic simplification.

```
1  Add0R[Equiv]: ⟦ ⟨Exp#⟩ + 0 ⟧ → Exp# ;
2  Add0L[Equiv]: ⟦ 0 + ⟨Exp#⟩ ⟧ → Exp# ;

3  Product0R[Equiv]: ⟦ ⟨Exp#⟩ * 0 ⟧ → ⟦0⟧ ;
4  Product0L[Equiv]: ⟦ 0 * ⟨Exp#⟩ ⟧ → ⟦0⟧ ;

5  Product1R[Equiv]: ⟦ ⟨Exp#⟩ * 1 ⟧ → Exp# ;
6  Product1L[Equiv]: ⟦ 1 * ⟨Exp#⟩ ⟧ → Exp# ;

7  Minus1[Equiv]:  ⟦ ⟨Exp#⟩ - 0 ⟧ → Exp# ;
8  Divide1[Equiv]: ⟦ ⟨Exp#⟩ / 1 ⟧ → Exp# ;
```

**5.1.3 Example.** Simplifying created function applications.

```
1  Beta[Equiv]: ⟦ [x1↦⟨Exp#2.x1⟩] ⟨Exp#1⟩ ⟧ → Exp#2[#1] ;
```

### 5.2 Constrained Equivalences

In most cases, however, an equivalence is constrained by an attribute. *To Be Done*... **TBD**

## 6 Running HACS

### 6.1 Limitations

- At most one `nested` declaration per `token`.
- It is not possible to use binders and left recursion in the same production (with same precedence).

# A Grammar Reference

We include details about the grammar here.

## A.1 Token Reference

Summary of tokens and regular expressions. . . *To Be Done*. . . **TBD**

```
sort RE | ⟦ ⟨REChoice*_"|"⟩ ⟧ ;

sort REChoice
| ⟦ nested ⟨RESimple⟩ ⟨RESimple⟩ ⟧
| ⟦ ⟨REUnit*⟩ ⟧
;

sort REUnit | ⟦ ⟨RESimple⟩ ⟨Repeat?⟩ ⟧ ;

sort Repeat | ⟦ ? ⟧
            | ⟦ * ⟧
            | ⟦ * _ ⟨String⟩ ⟧
            | ⟦ + ⟧
            | ⟦ + _ ⟨String⟩ ⟧
            ;

sort RESimple
| ⟦ ⟨String⟩ ⟧
| ⟦ ⟨Word⟩ ⟧
| ⟦ ⟨FragmentRef⟩ ⟧
| ⟦ . ⟧
| ⟦ [ ⟨REClass⟩ ] ⟧
| ⟦ ( ⟨RE⟩ ) ⟧
;

sort REClass
| ⟦ ^ ⟨REClassFirstChar⟩ ⟨RERangeEnd?⟩ ⟨RERange*⟩ ⟧
| ⟦ ⟨REClassFirstChar⟩ ⟨RERangeEnd?⟩ ⟨RERange*⟩ ⟧
;
sort RERange | ⟦ ⟨REClassChar⟩ - ⟨REClassChar⟩ ⟧ ;
sort RERangeEnd | ⟦ - ⟨REClassChar⟩ ⟧ | ;

token REClassFirstChar | [^^\n\\] | ⟨EscapedChar⟩ ;
token REClassChar | [^\]\n\\] | ⟨EscapedChar⟩ ;
```

## A.2 Precedence and Left Recursion

Some notes on the internals way HACS deals with precedence and left recursion.

*A.2.1 Remark* (parser precedence). The different precedence levels just become different low level productions, with the exception of left recursion, which is eliminated by rewriting rules. So, for example,

```
sort Exp | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1 | ⟦ ⟨Int⟩ ⟧@2 ;
```

really means

```
sort Exp | Exp@1 ;
```

```
2  sort Exp@1 | scheme Leftify[@2, Exp@1Tail]
3             | sugar ⟦ Exp@2 Exp@1Tail ⟧ → Leftify[#@2, Exp#@1Tail] ;

4  sort Exp@1Tail | sugar ⟦+ ⟨Exp#@2⟩⟧ Exp#@1Tail → Plus[#@2, Exp#@1Tail]
5                 | sugar ⟦⟧ → Done ;

6  Leftify[#1, Done] → #1 ;
7  Leftify[#1, Plus[#2, #3]] → Leftify[⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧, #3] ;

8  sort Exp@2 | Int ;
```

where `Leftify`, `Plus`, and `Done`, are internal symbols (so probably really named `Exp@1Tail@Leftify` *etc.*). The right to left recursion conversion happens with the `Leftify` rules, where the actual left recursive nested expressions are built by the second rule.

## A.3 "Raw" Terms

**A.3.1 Notation.** "Internal" or "raw" term syntax follows these conventions:

- Names containing a `#` are *meta-variables*. A meta-variable can be followed by a sequence of *meta-arguments* in `[]`s and is then called a *meta-application*. When used with concrete syntax, meta-variables have the form *Sort#i*.
- Names starting with a lower case letter are (plain) *variables*, which can be used as *binders* in constructions or as an *occurrence* raw term.
- All other names are *constructors*. A constructor followed by `()`s with a number of `,`-separated *scoped subterms* is called a *construction*. The scoped subterms have a very specific form: an optional sequence of variables in `[]`s sets up the binders for the scoped subterm and then the term itself is last. Construction is used for both data and defined schemes (the latter roughly corresponding to functions).

Notice that in raw form we do not have to use meta-variables with the same name as the sort – `#x` can be used as the name for the first `Exp`-sorted argument. The binding that is introduced, of `x` in each rule, is written as a raw `[]`d prefix to the scope subterm. Especially the `Var` rule is interesting: it shows how the same `x` variable name can be used in the raw form – the `[x]` – and inside parsed content – the ⟦x⟧ – the token management knows both are `Var` tokens that must thus be identified.[4]

## A.4 Complete Grammar

HACS is a HACS . . .

```
1  // $Id: hoacs.hx,v 3.1 2013/08/23 07:12:32 krisrose Exp $
2  module hoacs {

3  //// STRUCTURE.

4  sort Module | ⟦ module ⟨Identifier⟩ { ⟨Declaration*_[;]⟩ } ⟧ ;
5  sort Declaration | Module | LexicalDeclaration | SortDeclaration | Rule ;

6  //// LEXICAL DECLARATIONS.

7  sort LexicalDeclaration
8  | ⟦ space ⟨RegExp⟩ ⟧
9  | ⟦ token ⟨SortName⟩ | ⟨RegExp⟩ ⟧
10 | ⟦ token fragment ⟨Identifier⟩ | ⟨RegExp⟩ ⟧
```

[4] In this case it is a furher lucky coincidence that the lexical form of the token has a form that fits the HACS rules for a raw variable name but this can be repaired when necessary, as explained in Appendix A.

```
;
sort RegExp
| ⟦ ⟨RegExp@1*_[|]⟩ ⟧  // choice
| ⟦ ⟨RegExp@2*⟩ ⟧@1  // concatenation

| ⟦ ⟨RegExp@2⟩ ⟨Repeat?⟩ ⟧@1
| ⟦ ⟨STRING⟩ ⟧@2 | ⟦ ⟨CLASS⟩ ⟧@2 | ⟦ ⟨Identifier⟩ ⟧@2
| ⟦ ( ⟨RegExp⟩ ) ⟧@2 → RegExp
;
sort Repeat | ⟦* ⟨RepeatSep?⟩⟧ | ⟦+ ⟨RepeatSep?⟩⟧ | ⟦?⟧ ;
sort RepeatSep | "_" RegExpSimple ;

//// SORTS.

sort SortDeclaration

;

sort Sort | ⟦ ⟨SortRef⟩ ⟨SimpleSort*⟩ ⟧ ;
sort SortRef | META | VAR ;
sort SimpleSort | SortRef | ⟦ ( ⟨Sort⟩ ) ⟧ ;

sort SortAlternative
| ⟦ | ⟨Form⟩ ⟧
| ⟦ | scheme ⟨Form⟩ ⟧
| ⟦ | ⟨Form⟩ → ⟨Form⟩ ⟧
| ⟦ | ⟨AttributeForm⟩ ⟧
;

sort Form
| ⟦ ⟨CON⟩ [ ⟨ScopeSort*_[,]⟩ ] ⟧
| ⟦ ⟨CON⟩ ⟧ → ⟦ ⟨CON⟩[] ⟧
| FormUnit+
;
sort ScopeSort |  ⟦ ⟨SortName⟩ ⟧ ;
FormUnit | ⟦ ⟨SortPrec⟩ ⟨Repeat?⟩ ⟧ | ⟦ ⟦ ⟨ParsedFormUnit*⟩ ⟧ ⟨Repeat?⟩ ⟧ ;
ParsedFormUnit | ⟨PARSEDCHAR⟩ | ⟦ ⟨ ⟨ParsedScopeSort⟩ ⟩ ⟧ ;
ParsedScopeSort | ⟦ ⟨ScopePrefix*⟩ ⟨SortPrec⟩ ⟧ | ⟦ . ⟨SortPrec⟩ ⟧ | ⟦ ⟨SortPrec⟩ . ⟧ ;
ScopePrefix | ⟦ ⟨SortName⟩ . ⟧ ;
SortPrec | SortName | ⟦ ⟨SortName⟩ @ ⟨INTEGER⟩ ⟧ ;


AttributeForm | ⟦ ⟨ATTRIBUTEKIND⟩ ⟨AttributeName?⟩ : ⟨SortName?⟩ = ⟨SortName⟩ ⟧
                 | ⟦ ⟨ATTRIBUTEKIND⟩ ⟨AttributeName?⟩ : . ⟨SortName?⟩ = ⟨SortName⟩ ⟧ ;

SortName | META | ⟦⟨.VAR⟩⟧ ;

Identifier | META | VAR | CON ;
AttributeName | Identifier | Literal ;

//// RULES.
Rule | ⟦ ⟨RuleHead?⟩ ⟨Term⟩ → ⟨Term⟩ ⟧ ;
RuleHead | ⟦ ⟨CoreTerm⟩ : ⟧ ;

//// TERMS.
Term |
  ⟦ ⟨Term⟩ ⟨ATTRIBUTEKIND⟩ ⟨AttributeName⟩ { ⟨Simple⟩ : ⟨CoreTerm⟩ } ⟧
| ⟦ ⟨Term⟩ ⟨ATTRIBUTEKIND⟩ ⟨AttributeName⟩ { ⟨CoreTerm⟩ } ⟧
| ⟦ ⟨Term⟩ ⟨ATTRIBUTEKIND⟩ ⟨META⟩ ⟧
```

```
| CoreTerm
;
CoreTerm |
  ⟦ CON [ ⟨Scope*_[,]⟩ ] ⟧ | ⟨CON⟩⇒⟦⟨CON⟩[]⟧
| ⟦ ⟨META⟩ ⟨MetaArgument*⟩ ⟧
| ⟦ ⟦ ⟨ParsedUnit*⟩ ⟧ ⟧
| Simple
;
Scope | ⟦ ⟨VAR.⟩ . ⟨VAR.Scope⟩ ⟧ | Term ;
ParsedUnit | PARSEDCHAR | ⟦ ⟨ ⟨Term⟩ ⟩ ⟧ ;
Simple | ⟦ ⟨.VAR⟩ ⟧ | Literal ;
Literal | STRING | NUMBER ;
```

```
//// TOKENS.
token META | "" (UPPER|LOWER)* SUFFIX*   | "" STRING SUFFIX* ;
token VAR  | LOWER NAMECHAR* SUFFIX* | "v" STRING SUFFIX* ;
token CON  | UPPER NAMECHAR* SUFFIX* | SYMBOL* SUFFIX* | "C" STRING SUFFIX* ;
token fragment SUFFIX | "_" NAMECHAR+ | DIGIT+ ;

token STRING | ["] STRINGCHAR* ["] ;
token fragment STRINGCHAR | [¬\\"]
| [\\] [¬0-7Uu] | [\\] [0-7]? [0-7]? [0-7]
| [\\] [uU] [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-fA
;
token INTEGER | [-+]* DIGIT+ ;
token NUMBER | (INTEGER | [-+]* DIGIT* [.] DIGIT+) ( [Ee] [+-] DIGIT+ )? ;

token ATTRIBUTEKIND | [↑↓] ;

token fragment UPPER | [A-Z] ;
token fragment LOWER | [a-z] ;
token fragment DIGIT | [0-9] ;
token fragment NAMECHAR  | LOWER | UPPER | "-" ;
token fragment SYMBOL | [¬A-Za-z0-9\\-] ;

token CLASS | "[" "¬"? CLASSCHAR* "]" ;
token fragment CLASSCHAR | [¬\\\[\]] | [\\] (LOWER | [\\\[\]]) ;

token PARSEDCHAR | [¬[[]⟨⟩ ] ;


token LL | '\u27e6' ; // ⟦
token RR | '\u27e7' ; // ⟧
token L | '\u27e8' ; // ⟨
token R | '\u27e9' ; // ⟩
space [ ]+ | "//" [¬\n\r]* | nested '/*' '*/' ;

}
```

## References

[1] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[2] Kristoffer Rose. Combinatory reduction systems with extensions. SourceForge project `http://crsx.sf.net`, October 2012.