Layout-sensitive Language Extensibility with SugarHaskell

Sebastian Erdweg Felix Rieger Tillmann Rendel Klaus Ostermann
University of Marburg, Germany

Abstract

Programmers need convenient syntax to write elegant and concise programs. Consequently, the Haskell standard provides syntactic sugar for some scenarios (e.g., do notation for monadic code), authors of Haskell compilers provide syntactic sugar for more scenarios (e.g., arrow notation in GHC), and some Haskell programmers implement preprocessors for their individual needs (e.g., idiom brackets in SHE). But manually written preprocessors cannot scale: They are expensive, error-prone, and not composable. Most researchers and programmers therefore refrain from using the syntactic notations they need in actual Haskell programs, but only use them in documentation or papers. We present a syntactically extensible version of Haskell, SugarHaskell, that empowers ordinary programmers to implement and use custom syntactic sugar.

Building on our previous work on syntactic extensibility for Java, SugarHaskell integrates syntactic extensions as sugar libraries into Haskell's module system. Syntax extensions in SugarHaskell can declare arbitrary context-free and layout-sensitive syntax. SugarHaskell modules are compiled into Haskell modules and further processed by a Haskell compiler. We provide an Eclipse-based IDE for SugarHaskell that is extensible, too, and automatically provides syntax coloring for all syntax extensions imported into a module.

We have validated SugarHaskell with several case studies, including arrow notation (as implemented in GHC) and EBNF as a concise syntax for the declaration of algebraic data types with associated concrete syntax. EBNF declarations also show how to extend the extension mechanism itself: They introduce syntactic sugar for using the declared concrete syntax in other SugarHaskell modules.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Extensible languages; D.2.11 [*Software Architectures*]: Domain-specific architectures; D.2.13 [*Reusable Software*]

General Terms Languages, Design

Keywords SugarHaskell, Haskell, language extension, syntactic sugar, layout-sensitive parsing, DSL embedding, language composition, arrows, SugarJ

1. Introduction

Many papers on Haskell programming propose some form of syntactic sugar for Haskell. For instance, consider the following code excerpt from a paper about applicative functors [McBride and Paterson 2008]:

```
instance Traversable Tree where
  traverse f Leaf         = [| Leaf |]
  traverse f (Node l x r) =
    [| Node (traverse f l) (f x) (traverse f r) |]
```

The "idiom brackets" used in this listing are not supported by the actual Haskell compiler; rather, the paper explains that they are a shorthand notation for writing this:

```
instance Traversable Tree where
  traverse f Leaf         = pure Leaf
  traverse f (Node l x r) =
    pure Node <*> (traverse f l) <*> (f x) <*> (traverse f r)
```

Such syntactic sugar is quite common. Sometimes it is eventually supported by the compiler (such as *do* notation for monads); sometimes preprocessors are written to desugar the code to standard Haskell (such as the *Strathclyde Haskell Enhancement* preprocessor[1] which supports, among other notations, the idiom brackets mentioned above), and sometimes such notations are only used in papers but not in actual program texts. Extending a compiler or writing a preprocessor is hard, elaborate, and not modular, since independently developed compiler extensions or preprocessors are hard to compose.

Another practical problem of syntactic language extension is that the integrated development environment (IDE) should know how to deal with the new syntax, e.g., for syntax coloring, auto completion, or reference resolving. IDEs can be extended, of course, but this again is not a modular solution.

We propose a generic extension to Haskell, SugarHaskell, with which arbitrary syntax extensions can be defined, used, and composed as needed. In SugarHaskell, a syntactic extension is activated by importing a library which exports the syntax extension and defines a desugaring of the extension to SugarHaskell. Using SugarHaskell, the code for the example above looks like this:[2]

```
import Control.Applicative
import Control.Applicative.IdiomBrackets

instance Traversable Tree where
  traverse f Leaf         = (| Leaf |)
  traverse f (Node l x r) =
    (| Node (traverse f l) (f x) (traverse f r) |)
```

The syntactic extension and its desugaring is defined in the library IdiomBrackets. By importing this library, the notation and its desugaring are activated within the remainder of the file. When the SugarHaskell compiler is invoked, it will desugar the brackets to the code using pure and <*> from above. Files which do not import IdiomBrackets are not affected by the syntactic extension. If more than one syntax extension is required in the same file, the extensions are composed by importing all of them. Conflicts can

Haskell'12, September 13, 2012, Copenhagen, Denmark.

[1] `http://personal.cis.strath.ac.uk/conor.mcbride/pub/she`

[2] To avoid syntactic overlap with Template Haskell, we follow Strathclyde Haskell Enhancement and implement rounded idiom brackets.

arise if the extensions overlap syntactically, but this is rare for real-world examples and can usually be disambiguated easily [Erdweg et al. 2011b].

SugarHaskell also comes with an Eclipse-based development environment specifically tailored to support syntactic extensions. By importing the IdiomBrackets library, syntax coloring for the extended syntax is automatically provided. More advanced IDE services can be defined in and imported from *editor libraries* [Erdweg et al. 2011a].

It makes a significant difference that the target of the desugaring is SugarHaskell and not Haskell, because this means that the syntax extension mechanism is itself syntactically extensible. We will illustrate this issue with a case study that allows the definition of EBNF grammars in Haskell. Besides desugaring an EBNF grammar into an algebraic data type (the abstract syntax) and a Parsec parser (the concrete syntax), we generate yet another syntactic extension that enables using the concrete syntax in Haskell expressions and patterns directly.

The idea of library-based syntactic extensibility is not new. SugarHaskell builds on our earlier work on SugarJ, a syntactically extensible version of Java [Erdweg et al. 2011b,a]. The research contributions of this paper are as follows:

- SugarJ is tightly coupled to the Java programming language. To create SugarHaskell, we have decoupled syntax extension mechanism from the underlying programming language by creating an interface. We describe the design of this interface and how we used it to implement SugarHaskell.[3]
- Haskell presents a new technical challenge not present in Java: Layout-sensitive parsing [Marlow (editor) 2010, Sec. 2.7]. SugarHaskell allows the definition of layout-sensitive syntactic extensions and is, to the best of our knowledge, the first declaratively extensible parser for Haskell with layout-sensitive syntax. To realize layout-sensitive parsing, we have significantly extended a core technology on which SugarHaskell builds, namely the SDF formalism for syntax descriptions [Heering et al. 1989].

In addition to these research contributions, we believe that this work can also contribute very practically to the Haskell community. Haskell programmers often strive to express programs elegantly and concisely, using built-in features such as user-defined infix notation and layout-sensitive do notation. But since these built-in features are not always enough to express the desired syntax, Haskell compiler writers add language extensions to their compilers to support additional syntactic sugar. The Haskell community can benefit from SugarHaskell in two ways:

- SugarHaskell empowers ordinary library authors to provide appropriate notation for the use of their libraries without having to change a Haskell compiler.
- SugarHaskell assists language designers by providing a framework for prototyping and thorough experiments with language extensions that propose to change Haskell's syntax.

We show through a number of examples that it is simple and practical to implement a wide range of frequently desired syntactic extension in SugarHaskell.

Finally, to avoid confusion, within this paper we refrain from prettyfying code: All syntactic sugar is implemented with SugarHaskell and code is reproduced literally in the paper.

[3] SugarHaskell is open-source and available at `http://sugarj.org`.

```
exp ::= ...
      | proc pat -> cmd

cmd ::= exp -< exp
      | exp -<< exp
      | (| exp cmd ... cmd |)
      | cmd exp
      | cmd qop cmd
      | (cmd)
      | \ pat ... pat -> cmd
      | let decls in cmd
      | if exp then cmd else cmd
      | case exp of { calt; ...; calt }
      | do { cstmt; ...; cstmt }
```

Figure 1. Syntactic additions for arrow notation.

2. SugarHaskell by example

To illustrate SugarHaskell, let us integrate syntactic sugar for programming with arrows [Hughes 2000]. Arrows are a versatile generalization of monads and, like monads, arrows are somewhat cumbersome to use without syntactic support. For this reason, Paterson [2001] proposed *arrow notation* to make programming with arrows more convenient. In this section, we implement arrow notation with SugarHaskell.

We are not the first to support arrow notation for Haskell. Paterson developed a preprocessor[4] that translates Haskell code with arrow notation into Haskell 98 code. Furthermore, GHC supports arrow notation through a compiler extension, which can be activated by the `-XArrows` flag [GHC Team 2012, Section 7.13]. In contrast, SugarHaskell empowers regular programmers to integrate custom syntactic extensions that compose.

2.1 Arrow notation

Figure 1 summarizes the syntactic extension for arrow notation as specified by GHC [GHC Team 2012, Section 7.13]. First of all, arrow notation introduces new expression syntax `proc pat -> cmd` where `proc` is a new keyword for building arrows whose input matches `pat` and whose output is determined by the command `cmd`. Commands are like expressions but provide different syntax for applications. The first and second command productions specify arrow application where the right-hand-side expression is input to the arrow described by the left-hand-side expression. Here, GHC (and we) distinguish forwarding arrow application (`exp -< exp`) from the arrow application (`exp -<< exp`) that uses `app` from the `ArrowApply` type class. The third and fourth productions declare application of an expression to commands and vice versa. The brackets (|...|) have been introduced into GHC to syntactically distinguish these two forms of application.

An example SugarHaskell program that uses arrow notation is shown in Figure 2. It activates arrow notation by importing the arrow sugar library `Control.Arrow.Syntax` alongside the standard arrow library. Arrow notation is only active where the import is in scope, that is, in the current module. Therefore, it is possible to use competing syntactic extensions in different modules, but also to compose different syntax extensions in a single module. For example, idiom brackets (Section 1) do not conflict with arrow notation since brackets in arrow notation can only occur inside a command. Hence, these two sugar libraries can be used within the same module. Let us now look at the implementation of the arrow sugar library.

A sugar library consists of two artifacts: A grammar that specifies an extended syntax and a transformation that translates the

[4] `http://hackage.haskell.org/package/arrowp`

```
import Control.Arrow
import Control.Arrow.Syntax

eval :: (ArrowChoice a, ArrowApply a) =>
     Exp -> a [(Id, Val a)] (Val a)
eval (Var s) = proc env ->
       returnA -< fromJust (lookup s env)
eval (Add e1 e2) = proc env -> do
       ~(Num u) <- eval e1 -< env
       ~(Num v) <- eval e2 -< env
       returnA -< Num (u + v)
eval (If e1 e2 e3) = proc env -> do
       ~(Bl b) <- eval e1 -< env
       if b
         then eval e2 -< env
         else eval e3 -< env
eval (Lam x e) = proc env ->
       returnA -< Fun (proc v -> eval e -< (x,v):env)
eval (App e1 e2) = proc env -> do
       ~(Fun f) <- eval e1 -< env
       v <- eval e2 -< env
       f -<< v
```

Figure 2. Hughes's lambda-calculus interpreter [Hughes 2000] using arrow notation in SugarHaskell.

extended syntax into Haskell code (or Haskell code extended by other sugar libraries). To specify the syntax, we employ the generalized LR parsing formalism SDF [Heering et al. 1989], which we extended to support layout-sensitive languages. SDF has two major advantages over other parsing technologies. First, since it is a generalized LR parser, it supports declarative grammar specifications where we do not need to concern ourselves with left-recursion or encoding priorities. Second, SDF organizes grammars in composable modules and features a number of disambiguation mechanisms that make it possible to add syntax without changing previous syntax definitions [Erdweg et al. 2012a]. This enables us to modularly add syntactic extensions to Haskell without changing our Haskell grammar.

We have decomposed the syntax definition for arrow notation into three sugar libraries: one for command alternatives, one for command statements, and one for commands themselves. The latter one is shown in Figure 3. A SugarHaskell sugar library integrates into Haskell's module system. Accordingly, each sugar library starts with a module declaration and a list of import statements. These imports typically refer to other sugar libraries whose syntax is extended. The body of a sugar library is composed of SDF syntax declarations and desugaring transformations (more on desugarings later). Essentially, the syntax declaration in Figure 3 reflects the EBNF grammar from Figure 1. In SDF, the defined nonterminal appears on the right-hand side of the arrow `->`. Hence, the first production declares a new syntactic form for Haskell expressions. After a production, a list of annotations can follow in curly braces. The `cons` annotation specifies the name of the AST node corresponding to a production. The annotations `left` and `right` declare a production to be left-associative or right-associative, respectively. Finally, `longest-match` denotes that in case multiple parses are possible (SDF uses a generalized parser), the longest one should be chosen. These productions are supplemented with priority declarations (left out for brevity), which, for example, specify that the `ArrAppBin` production has precedence over the `ArrOpApp` production.

By importing the `Control.Arrow.Syntax.Command` module, a program using the extended syntax can already be parsed by SugarHaskell. However, compilation will fail because the parsed AST contains arrow-specific nodes like `ArrProcedure` that will not be understood by the compiler. Therefore, we require a desugaring transformation that relates the arrow-specific nodes to Haskell nodes (or nodes from another syntactic extension). To implement tree transformations, SugarHaskell employs the Stratego term-rewriting system [Visser 2001]. Stratego rules are based on pattern matching but, in contrast to many other systems, Stratego rules are open for extension: A rule can be amended in a separate module to handle more syntactic forms [Hemel et al. 2010]. This way, all SugarHaskell extensions contribute to a single desugaring transformation that desugars an AST bottom-up.

```
module Control.Arrow.Syntax.Command where

import Control.Arrow.Syntax.Alternatives
import Control.Arrow.Syntax.Statement

context-free syntax
  "proc" HaskellAPat "->" ArrCommand
      -> HaskellExp {cons("ArrProcedure")}

  HaskellExp "-<" HaskellExp
      -> ArrCommand {cons("ArrFirst")}

  HaskellExp "-<<" HaskellExp
      -> ArrCommand {cons("ArrHigher")}

  "(|" HaskellExp ArrCommand+ "|)"
      -> ArrCommand {cons("ArrForm")}

  ArrCommand HaskellExp
      -> ArrCommand {cons("ArrAppBin"), left}

  ArrCommand HaskellQop ArrCommand
      -> ArrCommand {cons("ArrOpApp"), right}

  "\\" HaskellFargs "->" ArrCommand
      -> ArrCommand {cons("ArrAbs")}

  "do" ArrStmtList
      -> ArrCommand {cons("ArrDo"), longest-match}

  ...
```

Figure 3. SugarHaskell syntax extension for arrow notation.

Figure 4 displays an excerpt of the desugaring transformation for arrow notation. First, let us inspect the import statements. The first import just brings the concrete and abstract command syntax into scope, which is the input language of the transformation we are about to define. However, the second import is special; it activates a SugarHaskell extension that does not affect the object language Haskell but the metalanguage Stratego. The sugar library `Meta.Concrete.Haskell` activates concrete syntax for transformations [Visser 2002], that is, it enables metaprogrammers to describe AST transformations by concrete syntax within |[...]| instead of abstract syntax. Since SugarHaskell extensions are self-applicable, syntactic extensions to the metalanguage can be expressed as a sugar library as well. Moreover, in our example, the metaextension is further extended by `Control.Arrow.Syntax.Concrete`, which enables concrete syntax for arrow commands after the `cmd` keyword.

Using concrete Haskell syntax in Stratego transformations, the desugaring transformation follows the GHC translation rules for arrow notation [Paterson and Peyton Jones 2004] except for some optimizations. The entry point of our desugaring is the `desugar-arrow` rule as declared by the **desugarings** block. Each Stratego rule declares a pattern on the left-hand side of the arrow `->` and produces the term on the right-hand side of the arrow. In concrete syntax, we use `$` to escape to the metalanguage in correspondence with TemplateHaskell [Sheard and Peyton Jones 2002]. Accordingly, in the first transformation rule `desugar-arrow` in Figure 4, the pattern matches on an arrow procedure and binds the Stratego variables `pat`

```
module Control.Arrow.Syntax.Desugar where

import Control.Arrow.Syntax.Command
import Meta.Concrete.Haskell
import Control.Arrow.Syntax.Concrete

desugarings
  desugar-arrow

rules
  desugar-arrow :
    |[ proc $pat -> $cmd ]| ->
    |[ arr (\$pat -> $(<tuple> vars))
       >>> $(<desugar-arrow'(|vars)> cmd) ]|
  where <free-pat-vars> pat => vars

  desugar-arrow'(|vars) :
    cmd |[ $f -< $e ]| ->
    |[ arr (\$(<tuple-pat> vars) -> $e) >>> $f ]|

  desugar-arrow'(|vars) :
    cmd |[ $f -<< $e ]| ->
    |[ arr (\$(<tuple-pat> vars) -> ($f, $e)) >>> app ]|

  desugar-arrow'(|vars) :
    cmd |[ do $c
              $*cs ]| ->
    |[ arr (\$(<tuple-pat> vars) ->
            ($(<tuple> vars), $(<tuple> vars)))
       >>> first $(<desugar-arrow'(|vars)> c)
       >>> arr snd
       >>> $(<desugar-arrow'(|vars)> cmd |[do $*cs]|) ]|

  ...
```

Figure 4. Desugaring transformation for arrow notation.

and cmd. If the matching succeeds, the rule produces a term that constructs an arrow with arr from a lambda expression and composes (>>>) this arrow with result of desugaring cmd. Note that in Stratego angled brackets <r> t denote an application of the rewrite rule r to the term t.

The module Control.Arrow.Syntax finally imports and reexports the two modules that define the syntax and desugaring for arrow notation. Since sugar libraries are integrated into Haskell's module system, an import statement suffices to activate the syntactic extension as illustrated in Figure 2. Moreover, SugarHaskell modules that contain (possibly sugared) Haskell code compile into a pure Haskell module. Therefore, SugarHaskell programs are interoperable with regular Haskell programs: The application of SugarHaskell in a library is transparent to clients of that library.

2.2 Layout-sensitive syntactic extensions

In order for a syntactic extension to integrate into Haskell nicely, the syntactic extension needs to adhere to the layout-sensitivity of Haskell code. For example, arrow notation includes arrow-specific do blocks that consists of a sequence of command statements, as visible in the interpreter in Figure 2 and the last production in Figure 3. All existing layout-sensitive languages we know of employ hand-tuned lexers or parsers. However, since we want regular programmers to write SugarHaskell extension, we need a declarative formalism to specify layout-sensitive syntax.

To this end, we have developed a variant of SDF that supports layout-sensitive languages. In our variant, SugarHaskell programmers can annotate productions with *layout constraints* that restrict the context in which this production may be used. Figure 5 shows the use of layout constraints in the definition of arrow-specific statement lists. In the figure, we have emphasized the layout-specific additions we made to SDF. A statement list can employ implicit or explicit layout. In the latter case, the statement list is encapsulated in curly braces and statements are separated by semicolons. Hence, an explicit statement list does not pose any layout constraints. What is more, an explicit statement list may even violate constraints imposed by the surrounding context. For example, the following is a syntactically valid Haskell program where the do block consists of three statements:

```
foo = do
  x <- foo
  let
{ y = bar x
; z = baz z }
  bac z
```

In SugarHaskell, such layout behavior is declared by the ignore-layout annotation.

Statement lists with implicit layout are harder to realize. Essentially, they need to adhere to two invariants. First, each statement may only extend to the right, that is, every token is further indented than the token that starts the statement. This invariant is expressed by the first constraint in Figure 5: 1.first.col selects the column of the starting token of the first subtree of the current production; in contrast, 1.left.col selects the column of the leftmost non-starting token of the first subtree of the current production. The second invariant declares that each statement in a statement list must start on the same column. This invariant is expressed by the second constraint on the last line of Figure 5.

More technical details on our layout-sensitive parser follow in Section 3.3. For now, let us point out that our layout-sensitive parser is not limited to the object language. We employ the same parser for parsing metaprograms, which thus can make use of layout-sensitive syntax. In particular, when using concrete Haskell syntax to declare transformations, the Haskell syntax is layout-sensitive. For example, the last rule of Figure 4 matches on an arrow-specific do block. The Haskell snippet used to match on such expressions is parsed layout-sensitively, that is, indenting or dedenting the remaining statement list $*cs will lead to a parse error. While this may seem overkill for such a small code snippet, it becomes essential when generating code that nests let, do, case, and where blocks.

```
module Control.Arrow.Syntax.Statement where

context-free syntax
  "let" HaskellDecIbinds -> ArrStmt {cons("ArrLetStmt")}
  HaskellPat "<-" ArrCommand -> ArrStmt {cons("ArrBind")}
  ArrCommand -> ArrStmt {cons("ArrCmdStmt")}

context-free syntax
  ArrImplStmtList  -> ArrStmtList {cons("ArrStmtList")}
  "{" ArrExplStmtList "}"
      -> ArrStmtList {cons("ArrStmtList"), ignore-layout}

  ArrStmt -> ArrExplStmtList
  ArrStmt ";" ArrExplStmtList
      -> ArrExplStmtList {cons("ArrStmtSeq")}

  ArrStmt -> ArrImplStmt {layout("1.first.col < 1.left.col")}
  ArrImplStmt -> ArrImplStmtList
  ArrImplStmt ArrImplStmtList -> ArrImplStmtList
   {cons("ArrStmtSeq"), layout("1.first.col == 2.first.col")}
```

Figure 5. SugarHaskell's layout constraints restrict the context in which a production may be used.

3. Technical realization

We realized SugarHaskell on top of our previous work on SugarJ [Erdweg et al. 2011b,a]. Like SugarHaskell, SugarJ is a syntactically extensible programming language that integrates syntactic extensions into the module system of the host language, that is, Java. To realize SugarHaskell, we significantly reengineered the SugarJ compiler to factor out host-language-specific components and to hide them behind an abstract data type. This way it becomes relatively easy to make additional languages syntactically extensible.

3.1 Background on SugarJ

The SugarJ compiler processes a source file by first *parsing* it into an AST, then *desugaring* the AST into an AST that contains no syntactic extensions, and finally *compiling* the desugared program. However, since in SugarJ syntactic language extensions are integrated into the module system of the host language, the SugarJ compiler needs to support two particular features: First, to react to a sugar-library import, the compiler needs to understand the module-relevant structure of source files. Second, to activate a sugar library dynamically, the compiler needs to be able to adapt the parser and desugaring transformation while processing a source file.

We realized the first requirement by incorporating knowledge about the relevant AST nodes into the compiler, so that the compiler recognizes ASTs and can react appropriately. For example, when the compiler encounters an import statement, it inspects the imported library to determine whether it is a regular library or a sugar library. If the library is a sugar library, the compiler activates it right away by adapting the parser and desugaring transformation.

To realize the second requirement, the compiler processes source files incrementally. It dissects any source file into a sequence of top-level entities that it parses, desugars, and compiles one after another. Examples of top-level entities in Java include package declarations, import statements, class declarations, and sugar declarations. For Haskell, we recognize module declarations, import statements, and the body of a module as top-level entities. To handle a source file incrementally, the compiler repeatedly parses the next top-level entity as an AST and the remainder of the file as a character string. It then desugars the parsed top-level entity, stores it for compilation, and possibly adapts the parser and desugaring transformation for the next iteration. Hence, the syntax of a SugarJ program can change after any top-level entity. For more details, we refer the reader to our prior work [Erdweg et al. 2011b].

3.2 The Haskell language library

We reengineered the SugarJ compiler to support host languages other than Java. To this end, we designed an abstract data type LanguageLib that encapsulates host-language specific components of the compiler. To date, we have implemented three instances of LanguageLib: JavaLib, HaskellLib, and PrologLib [Rieger 2012].

The important categories of abstract methods in LanguageLib are:

- *Initialization*, which comprises methods that set up the initial grammar, desugaring transformation, and editor services for the sugared language. For SugarHaskell, the initial grammar consists of full Haskell amended with SDF and Stratego grammars for specifying sugar libraries.
- *AST predicates*, which comprises methods to reflect on the parsed top-level entity. Each concrete language library needs to distinguish declarations of a module or namespace, import statements, language-specific entities, sugar libraries, and editor services. The SugarJ compiler uses these AST predicates to dispatch on the parsed AST.
- *Host-language processing*, which comprises methods to process host-language code. In particular, LanguageLib requires methods for processing a module declaration, import statements, and a module's body. The standard way of implementing these methods is to generate a host-language source file that contains pretty prints of the host-language entities. In addition, LanguageLib requires a method that compiles the generated source file.

Notably, the SugarJ compiler handles declarations of sugar libraries and editor services independent of concrete language libraries. Moreover, a language library can perform static checking and notify the programmer at compile time. For example, HaskellLib ensures that imports of Haskell modules are resolvable by calling ghc-pkg.

3.3 Layout-sensitive generalized parsing

Layout-sensitive languages typically do not belong to the class of context-free languages because counting and comparing indentation is required. Therefore, due to the context-sensitive nature of layout-sensitive languages, off-the-shelf parsers are not applicable and efficiency cannot be guaranteed.

We have developed a layout-sensitive variant of generalized LR parsing where layout constraints are declared as part of a grammar and restrict the valid applications of a production [Erdweg et al. 2012b].[5] Conceptually, we ignore layout at parse time and filter the resulting parse forest by enforcing the layout constraints at disambiguation time. However, the number of ambiguities is overwhelming so that this naive approach fails for performance reasons. To improve efficiency, we identified a subclass of layout constraints that in fact is context-free and can be enforced at parse time. In particular, layout constraints that only use the first(...) node selector can safely be enforced at parse time. For example, in accordance with the constraint on the last line of Figure 5, our parser never considers an ArrImplStmtList where the head and the tail of the list do not start on the same column.

We have evaluated our parser on all of Hackage. Results are promising: Of the 24 219 files that the *haskell-src-exts* parser could parse with a small, fixed selection of extensions, our parser was able to successfully parse 94 percent (resulting in the same parse tree as with explicit layout) with a median parse time of 17ms. We sampled the remaining files and found the following errors: First, we reject statements that start with a block comment on the same line since the comment is ignored and the statement appears to be indented too far (5 files). Second, our parser timed out after 30 seconds (40 files). Third, the *haskell-src-exts* parser wrongly ignores the language option *NondecreasingIndentation* whereas we enforce non-decreasing indentation (274 files). Finally, our parser failed to parse 1651 files even with explicit layout. Since this is independent of layout, we suspect inaccuracies in the SDF Haskell grammar, which we adapted from the Haskell transformation framework HSX[6] to feature layout constraints. In summary, these results suggest that our parser is working correctly for the majority of all files, but our test framework and the Haskell SDF grammar require a bit more work.

3.4 IDE support and static analyses

The SugarJ system also comes with Eclipse-based editor support [Erdweg et al. 2011a]. However, in contrast to other language frameworks, SugarJ's editor services are not predetermined. Instead, a programmer can declare custom editor services in *edi-*

[5] The implementation and raw evaluation data is open-source and available at https://github.com/seba--/layout-parsing.

[6] http://strategoxt.org/Stratego/HSX

tor libraries, which are integrated into the host-language module system. This way programmers can declare domain-specific syntax coloring, code folding, outlining, content completion, reference resolving, and hover help. Like any other library, an import statement brings an editor library into scope of a module and activates the contained editor services. These editor services match on extension-specific parts of the AST and provide according tool support. Since editor services of independent syntactic extensions match on independent parts of the AST, editor services compose [Erdweg et al. 2011a].

Similarly, a programmer can declare static analyses within a library. A static analysis is written in Stratego and matches on the non-desugared AST to produce a list of errors. Essentially, SugarJ supports a form of pluggable type system [Bracha 2004] where language designers can formalize and enforce domain-specific language invariants. In combination with syntactic sugar for metalanguages, this enables the definition of domain-specific type systems such as XML Schema, which we implemented in prior work [Erdweg et al. 2011b].

Since SugarHaskell is built on top of SugarJ, the same IDE integration and pluggable type system are available for SugarHaskell programmers. However, editor services and static analyses are not the focus of this work.

4. Case study

To demonstrate the flexibility and usefulness of SugarHaskell, we implemented a sugar library that extends Haskell with a DSL for syntax declarations, namely EBNF. A Haskell programmer can use this extension to specify an EBNF grammar, which we desugar into an algebraic data type (the abstract syntax) and Haskell functions to parse a concrete-syntax string into instances of that data type. Moreover, from a concrete EBNF grammar we generate yet another syntactic extension that allows programmers to use their own concrete syntax in Haskell code to pattern-match or construct values of their abstract syntax (the generated data type).

This case study particularly highlights two features of SugarHaskell. First, syntax extensions can go beyond simple syntactic sugar to increase the expressivity of a language. Second, the extension mechanism of SugarHaskell is self-applicable, that is, syntactic extensions can desugar into definitions of further syntactic extensions. Consequently, SugarHaskell supports an unlimited number of metalevels.

4.1 EBNF: A DSL for syntax declarations

Haskell's declarative nature and expressivity make it a good platform for experimenting with the design and implementation of other programming languages. For example, it is comparatively easy to write interpreters or type checkers in Haskell. However, in our own experience, experimentation and testing are often limited by the format in which example programs have to be fed into the interpreter, that is, as instances of an algebraic data type. Consequently, programmers experiment with their interpreter or type checker only on a small number of examples of very limited size.

To make writing examples easier, one could implement a parser. However, writing parsers is tedious and distracting. For that reason, we propose a syntactic integration of EBNF with which programmers can simultaneously declare the abstract and concrete syntax of the language under design. For example, Figure 6 shows a SugarHaskell program that specifies the concrete and abstract syntax of the lambda calculus using our EBNF embedding.

EBNF grammars are organized by nonterminal. For the lambda calculus, we use three nonterminals Var, Exp, and String, where String is primitive and describes sequences of non-whitespace characters. The concrete syntax of all other nonterminals is user-supplied. In addition to concrete syntax, a programmer specifies

```
module Lambda.Syntax where

import Data.EBNF.Syntax
import Data.EBNF.Data
import Data.EBNF.Parser

Var ::= String {Var}

Exp ::= Var                          {EVar}
      | "(" Exp Exp ")"              {EApp}
      | "lambda" Var "." Exp         {EAbs}
      | "(" Exp ")"
```

Figure 6. Declaration of concrete and abstract syntax of the lambda calculus using the EBNF sugar library.

abstract syntax by supplying the names of AST nodes in curly braces. If no node name is supplied, the corresponding production only forwards its children to the surrounding production but does not produce an AST node itself. For example, according to the lambda-calculus grammar, the string "lambda f. lambda x. (f x)" is concrete syntax for:

```
EAbs(Var "f", EAbs(Var "x", EApp(EVar(Var "f"), EVar(Var "x"))))
```

We desugar an EBNF grammar into multiple artifacts. First, to represent the abstract syntax, an EBNF grammar desugars into an algebraic data type using the following translation scheme:

EBNF	Haskell
nonterminal definition	data-type declaration
alternative with AST node name	constructor
nonterminal in concrete syntax	constructor field

Accordingly, the grammar from Figure 6 desugars into the following data-type declarations:

```
data Var = Var String
data Exp = EVar Var
         | EApp Exp Exp
         | EAbs Var Exp
```

To encode the concrete syntax of an EBNF grammar, we generate the definition of a Haskell function that parses a string into instances of the previous data types. The generated functions employ Parsec [Leijen and Meijer 2001] to parse the input and are used to derive an instance of the Read type class. Hence, the following declarations are generated for the lambda-calculus grammar:

```
parseVar :: ParsecT String Identity Var
parseVar = ...
instance Read Var where
  readsPrec _ input = ... runParser parseVar ...

parseExp :: ParsecT String Identity Exp
parseExp = ... (parseVar >>= return . EVar) <|> ...
instance Read Exp where
  readsPrec _ input = ... runParser parseExp ...
```

By generating a Parsec parser from EBNF, we also inherit Parsec's limitations: The parser of a left-recursive EBNF grammar will not terminate and if multiple productions are applicable, the parser always uses the first one and completely ignores the others. We address these problems in two ways. First, we implemented a domain-specific static analysis in SugarHaskell (cf. Section 3.4) that approximates whether an EBNF grammar is left-recursive and issues a domain-specific error message to the programmer if that is the case. Second, in the generated parser, we prefer productions that start with a keyword matching the input. The resulting parser

can be used to describe example lambda-calculus expressions in concrete syntax:

```
ident = read "lambda x. x" :: Exp
app = read "lambda f. lambda x. (f x)" :: Exp
```

We have designed the EBNF sugar library such that clients can configure which artifacts to generate from the grammar. To this end, the main desugaring of EBNF calls a fixed set of pattern-matching Stratego rules, each of which supports no input at all and always fails. Stratego's extensibility mechanism allows programmers to amend those rules in other modules to handle further input (a rule is only executed once even if definitions overlap) [Hemel et al. 2010]. Thus, by bringing further sugar libraries into scope, a programmer can effectively configure the desugaring of an EBNF grammar. This design is visible in Figure 6, where we activate the desugaring into data-type and parser declarations through the imports of Data and Parser, respectively. If we do not want a parser, we can drop the corresponding import to deactivate its generation. On the other hand, it is not possible to only deactivate the data-type generation because the generated parser depends on it. Hence, Parser reexports Data and an import of Parser activates Data as well. In addition to Data and Parser, a client of the EBNF sugar library can import Data.EBNF.MetaSyntax to activate a desugaring that employs SugarHaskell's self-applicability as we explain in the following subsection.

4.2 EBNF: A meta-DSL

The EBNF sugar library allows programmers to simultaneously define concrete and abstract syntax. Programmers can use the generated Parsec parser to declare example programs of their language in concrete syntax, which the parser translates into instances of the generated algebraic data type. However, in a syntactically extensible programming language like SugarHaskell such indirection is unnecessary—the example program could be parsed at compile time. Moreover, the generated Parsec parser does not allow programmers to use their concrete syntax for building compound ASTs such as EAbs (Var "x") (EApp ident (EVar (Var "x"))) or for pattern matching on ASTs.

To address these concerns, we provide another desugaring of EBNF grammars defined in Data.EBNF.MetaSyntax. This desugaring generates a syntactic extension of Haskell specific to a concrete EBNF grammar. To illustrate the generated sugar, Figure 7 displays a definition of the small-step operational semantics of the lambda calculus.

The function reduce realizes the reduction relation using concrete lambda-calculus syntax in pattern matching and data construction. Concrete syntax is wrapped in brackets |[...]| to distinguish it from regular Haskell code. Within concrete syntax, $ can be used to escape to the metalanguage, that is, Haskell. Accordingly, in the first equation of reduce, the pattern |[((**lambda** $v. $b) $e)]| corresponds to the Haskell pattern (EApp (EAbs v b) e) that binds the pattern variables v, b, and e. Similarly, on the right-hand side of the second equation of reduce, concrete syntax is used to produce a new lambda-calculus expression: |[($(reduce e1) $e2)]| corresponds to the Haskell expression EApp (reduce e1) e2.

As visible in the last equation of reduce, MetaSyntax also incorporates some disambiguation mechanisms. The problem is that a pattern |[$v]| can be understood in different ways. It could either refer to a variable v, to an expression v, or to an expression variable (EVar v). Therefore, programmers can denote the syntactic category a concrete-syntax expression belongs to as |[Exp | ...]|, which rules out the first interpretation of |[$v]|. To distinguish the remaining possibilities, a programmer can also declare which syntactic category an escaped metaexpression belongs to. Hence, Var$ prefixes a metaexpression that describes a Var instance, whereas Exp$ prefixes an Exp expression.

```
module Lambda.Eval where

import Lambda.Syntax

reduce |[ ((lambda $v. $b) $e) ]|
  | isVal e = subst v e b
reduce |[ ($e1 $e2) ]|
  | not (isVal e1) = |[ ($(reduce e1) $e2) ]|
  | not (isVal e2) = |[ ($e1 $(reduce e2)) ]|
reduce |[ Exp | Var$v ]| = error ("free variable " ++ show v)

isVal |[ lambda $v. $e ]| = True
isVal _ = False

eval e
  | isVal e    = e
  | otherwise = eval (reduce e)

app = |[ lambda f. lambda x. (f x) ]|
ident = |[ lambda x. x ]|
identEta = |[ lambda x. ($ident x) ]|
```

Figure 7. Small-step operational semantics of the lambda calculus using MetaSyntax.

Technically, MetaSyntax desugars an EBNF grammar into a syntactic extension of Haskell. It produces productions that describe the concrete syntax in SDF

```
context-free syntax
  MSVar                          -> MSExp {cons("MS-EVar")}
  "(" MSExp MSExp ")"            -> MSExp {cons("MS-EApp")}
  "lambda" MSVar "." MSExp -> MSExp {cons("MS-EAbs")}
  "(" MSExp ")"                  -> MSExp {cons("NoConstr")}
```

as well as SDF productions that describe the integration into Haskell syntax:

```
context-free syntax
  "|[" MSExp "]|" -> HaskellExp   {cons("ToHaskellExp")}
  "|[" MSExp "]|" -> HaskellAPat {cons("ToHaskellAPat")}
  "$" HaskellExp  -> MSExp        {cons("FromHaskellExp")}
```

In addition, MetaSyntax provides a generic desugaring that translates concrete-syntax expressions into Haskell expressions. For example, this desugaring translates the AST of identEta in Figure 7

```
ToHaskellExp(MS-EAbs(MS-Var("x"),MS-EApp(
  FromHaskellExp(HSVar("ident")),
  MS-EVar(MS-Var("x")))))
```

into the corresponding Haskell expression:

```
EAbs (Var "x") (EApp ident (EVar (Var "x")))
```

Like all other desugarings in SugarHaskell, this translation is performed at compile time; there is no runtime overhead.

The essential feature of SugarHaskell, which also separates it from most other syntax extenders, is the self-applicability of the extension mechanism: Sugar libraries can declare syntactic sugar for defining further sugar libraries. In particular, EBNF can be seen as a DSL for declaring further user-specific language extensions. Therefore, we call such a language a meta-DSL [Erdweg et al. 2011b], that is, a DSL for defining DSLs.

5. Discussion and future work

The major goal of SugarHaskell is to support Haskell programmers in writing elegant and concise programs. In this section, we reflect on the practical advantages and limitations of using SugarHaskell.

5.1 Haskell integration

When proposing an extension of an existing system, it is important to ensure interoperability between the extended and the original system. SugarHaskell provides interoperability with Haskell by (1) forwarding valid Haskell programs unchanged (except for parsing and pretty printing) to GHC, (2) not relying on runtime support, (3) using the GHC package database to locate imported modules and (4) organizing and linking compiled files such that they can be used both with SugarHaskell and GHC, where GHC simply ignores any generated grammars and desugaring rules. Together, this supports the following interoperation scenarios:

- A Haskell program is compiled by SugarHaskell. This is supported because pure Haskell programs are forwarded unchanged to GHC.
- A Haskell library is used in a SugarHaskell program. This is supported because SugarHaskell uses the GHC package database to locate the Haskell library.
- A SugarHaskell library is used in a Haskell program. This is supported because extensions are just syntactic sugar: SugarHaskell programs always desugar into pure Haskell programs and no special runtime support is required. The library author would use SugarHaskell to create a Haskell version of its library and distribute that version to its users, who treat it like a usual Haskell library.

Hence, programmers can transparently employ SugarHaskell without letting their clients know.

Currently, SugarHaskell is not integrated in the Cabal build system or the ghci interactive Haskell interpreter. In our future work, we want to investigate whether such integration with cabal or ghci is feasible. The following scenarios would be worthwhile to enable:

- SugarHaskell programmers build SugarHaskell programs with Cabal.
- SugarHaskell programmers distribute SugarHaskell packages with Cabal and HackageDB.
- SugarHaskell programmers download, compile and install SugarHaskell packages from Hackage with cabal-install.
- Haskell programmers download, compile and install SugarHaskell packages from Hackage with cabal-install. This means that the packages on Hackage need to contain the generated Haskell files.
- SugarHaskell programmers can import sugar libraries and use syntactic sugar from the ghci prompt.
- SugarHaskell programmers can debug desugarings from the ghci prompt.

This integration would go beyond the current state of the art of preprocessor integration into the Haskell ecosystem. While Cabal supports preprocessors, it cannot track whether a preprocessor is available on the user's system. Preprocessors are therefore not automatically installed by cabal-install. SugarHaskell libraries, however, would be tracked as ordinary package dependencies.

5.2 Extension composition

SugarHaskell promotes extension composition by making it simple for programmers to use multiple extensions: A programmer just imports all corresponding sugar libraries. Therefore, it is important that SugarHaskell extensions actually can be used jointly, that is, it is important that sugar libraries compose.

In general, SugarHaskell inherits much of its composability support from the metalanguages it employs, namely SDF and Stratego. More specifically, SugarHaskell supports the composition of sugar libraries that are syntactically unambiguous, which is the common case [Erdweg et al. 2011b]. Such sugar libraries provide productions that extend different parts of the language or extend the same part with different syntax. Furthermore, since desugaring transformations typically only translate a sugar library's new syntax, there is no conflict between desugaring transformations of independent sugar libraries. All sugar libraries presented in this paper (idiom brackets, arrow notation, EBNF, EBNF metasyntax) are syntactically unambiguous and can be easily used within the same module.

In case two sugar libraries overlap syntactically, programmers can often use one of the disambiguation mechanisms of SDF [Heering et al. 1989]. For example, priorities declare precedence of one production over another, whereas reject productions can be used to restrict what can be parsed by a nonterminal. For example, we used reject productions

```
lexical syntax
  "proc" -> HaskellVARID {reject}
  "-<" -> HaskellVARSYM {reject}
  "-<<" -> HaskellVARSYM {reject}
```

in the arrow-notation sugar library to disallow the use of `proc` as a variable name and to reserve `-<` and `-<<` for arrow notation. Similarly, a programmer can disambiguate two conflicting sugar libraries by adding a third sugar library that applies SDF disambiguation mechanisms. There is no need to alter previously defined productions [Erdweg et al. 2012a].

5.3 Transformation language

In SugarHaskell, we employ Stratego as metalanguage for term transformation. From a language-design point of view, this is unattractive because it lacks regularity: The metalanguage is different from the object language. It would be more appealing to use the same language and language extensions on all metalevels.

However, we use Stratego for a good reason. As previously discussed in Section 2, the definition of a single Stratego rule can be separated into multiple equations that are located in different modules. Essentially, each equation corresponds to a pattern-matching case that can fail or succeed. When applying a transformation rule, Stratego tries each equation currently *in scope* until one succeeds or all have failed [Visser 2001; Hemel et al. 2010]. SugarHaskell makes heavy use of this extensibility mechanism.

In particular, all sugar libraries contribute to a single Stratego rule `desugar` through the declaration of **desugarings**. Whenever a programmer activates another sugar library using an import, one or more additional equations for `desugar` come into scope. SugarHaskell applies the single resulting desugaring transformation `desugar` to an AST bottom-up until a fixed point is reached. Hence, a sugar library can also desugar into an AST that another sugar library handles.

5.4 Hygiene

Hygienic transformations enable the transparent use of names in code transformations and avoid two potential conflicts [Clinger and Rees 1991]. First, when generating code that refers to a variable, this variable may not be captured at the transformation's call site. Instead, the variable must be resolved in the context of the transformation's definition. For example, the `IdiomBrackets` sugar library from Section 1 generates references to `pure` and `(<*>)`. This should be transparent to users of the sugar library and should not interfere with local declarations of functions of the same name. Second, a name capture can occur when a transformation introduces new variable bindings. These bindings may not capture any variables at the transformation's call site.

SugarHaskell does not support hygienic transformations. Hence, sugar libraries may produce accidental name capture. However,

we employ the convention of fully qualified names, which at least avoids most potential naming conflicts of the first category. For example, in the IdiomBrackets sugar library from Section 1, we in fact generate references to Control.Applicative.pure and Control.Applicative.(<*>) as well as a qualified import of Control.Applicative. In our experience, this convention makes unhygienic transformations much less harmful.

However, a clean solution to hygiene is desirable. Unfortunately, we cannot directly apply solutions to hygiene known from macro systems such as Scheme [Sperber et al. 2009; Clinger and Rees 1991]. The reason is that we want to support user-defined binding mechanisms that do not necessarily translate into a binding in the generated code. Therefore, we cannot infer variable scoping in the sugared syntax from the desugaring. Moreover, since we pretty print and compile regular Haskell code, we cannot enhance identifiers with context information; ultimately, each identifier must be represented as a simple string. We plan to investigate these issues in our future work.

5.5 Type-awareness

The preprocessor nature of SugarHaskell becomes most apparent when considering type-system integration and error reporting. While SugarHaskell supports user-defined static analyses before desugaring (see Section 3.4), these analyses are independent of Haskell's type system. SugarHaskell delegates actual type checking of desugared code to GHC, which consequently reports errors in terms of generated code. We see the following potential use cases of a tighter integration of type checking into SugarHaskell:

- Sugar libraries could declare extension-specific error messages in case the generated code fails to type-check. One interesting avenue of future work is to analyze the applicability of type-inference instrumentation [Heeren et al. 2003] to achieve extension-specific error messages.
- Type-dependent transformations could be used to generate specialized code for input of certain types, for example, to increase efficiency or to circumvent run-time ad-hoc polymorphism.
- Type-based syntax disambiguation [Bravenboer et al. 2005] could be used to select a parse tree in case there is a syntactic ambiguity. For example, arrow notation would not need a separate syntactic category command since arrows can be distinguished by type. Similarly, the EBNF metasyntax disambiguation |[Exp | ...]| would often be unnecessary because the expected syntactic category is implied by the expected type.

One interesting line of research that would enable these use cases is to feature type checking itself as an extensible component inside SugarHaskell. Instead of checking code generated from a sugar library, the sugar library could declare a type-system extension that defines new type rules for the added syntax. For example, the arrow-notation sugar library would declare type rules for checking the well-typedness of commands. In such system, all error checking and all error reporting should be in terms of original source code. However, there are many open research questions such as how can we ensure that extensions retain the invariants of the original type system? Further investigation is pending.

6. Related work

Syntactic extensibility has been the focus of researchers for a long time, from macro processors [McIlroy 1960; Layzell 1985; Tobin-Hochstadt et al. 2011], to attribute grammars [Knuth 1968; Van Wyk et al. 2010], extensible compiler frameworks [Ekman and Hedin 2007; Nystrom et al. 2003], and language workbenches [Kats and Visser 2010; Voelter and Solomatov 2010]. In prior work, we discussed the differences between these approaches and our framework in detail [Erdweg et al. 2012a, 2011b]. In a nutshell, we are different from most other approaches because our syntactic extension can use the full class of context-free languages, our extensions compose, and our extensibility mechanism is self-applicable. Here, we focus on related work that is more specific to Haskell.

6.1 TemplateHaskell

GHC supports compile-time metaprogramming with the *TemplateHaskell* language extension [Sheard and Peyton Jones 2002]. TemplateHaskell supports arbitrary compile-time computation via TemplateHaskell macros. They are written in Haskell and invoked explicitly with a special call syntax $(...). Macros can only be called in a fixed set of syntactical contexts. TemplateHaskell is tightly integrated with GHC, and macros can even access GHC's typing environment to analyze the program currently being compiled. TemplateHaskell is therefore not available for other Haskell compilers.

SugarHaskell also supports compile-time computation in the form of desugarings, but desugarings are written in Stratego and invoked implicitly whenever they are in scope. Desugarings can match on any constructors in the AST and even on constructors that have been introduced by other sugar libraries. SugarHaskell is independent of any specific Haskell compiler, but therefore also does not integrate into a compiler's typing environment. It would be interesting to implement part of the static analysis of a Haskell program, for example, name resolution, as a sugar library in Stratego to support more TemplateHaskell programming patterns in SugarHaskell.

GHC also supports a limited form of syntax extension via *quasiquotation* [Mainland 2007]. Syntax extensions are specified by writing a quasiquoter in Haskell, that is, essentially a standalone TemplateHaskell macro of type String -> Q Exp. The new syntax is used by explicitly invoking the quasiquoter with special call syntax [foo|...|]. The part ... can be arbitrary text and is processed by the quasiquoter foo at compile time. Quasiquotation is only available in a fixed set of syntactical contexts. Quasiquotation nests badly, because the outer quasiquoter would need to implement the quasiquotation mechanism manually in order to correctly handle the inner quasiquoter.

SugarHaskell's support for syntax extension is more declarative, because it is based on grammar rules instead of hand-written parsers. This means that SugarHaskell extensions compose better, since Sugar libraries can extend all parts of the base Haskell syntax as well as syntax introduced by other sugar libraries. In particular, nesting works out of the box without extra effort by the implementors of sugar libraries.

6.2 Preprocessors

The Haskell toolbox contains numerous preprocessors. The Haskell platform[7], a collection of blessed Haskell libraries and developer tools, includes the following preprocessors: the parser generator Happy[8], the lexer generator Alex[9], and hsc2hs[10], a generator for bindings to C functions. The Haskell Common Architecture for Building Applications and Libraries (Cabal)[11], the most common build and distribution system for Haskell, additionally supports

[7] http://hackage.haskell.org/platform/

[8] http://www.haskell.org/happy/

[9] http://www.haskell.org/alex/

[10] http://www.haskell.org/ghc/docs/latest/html/users_guide/hsc2hs.html

[11] http://www.haskell.org/cabal/

two other binding generators (c2hs[12] and greencard[13]) as well as cpphs[14], a reimplementation of the C preprocessor with better support for Haskell's lexical syntax. The standard C preprocessor is directly supported by GHC and, on Windows, even distributed with GHC.

The Strathclyde Haskell Enhancement (SHE)[15] is a handwritten preprocessor for Haskell. It is not based on a complete layout-sensitive Haskell parser, but on a lexer with layout heuristics. We have modeled our implementation of idiom brackets after SHE's implementation.

These and similar tools play two important roles in the Haskell ecosystem: (1) They extend the Haskell language with additional special-purpose constructs that are very useful for some applications, but not generally useful enough to warrant inclusion in the Haskell standard. (2) They allow language designers to provide prototype implementations of language extensions to the community. Unfortunately, it is impossible to compose these preprocessors to extend an extended language further. For example, it is not possible to use SHE to enable idiom brackets in the parser actions in a Happy parser because SHE does not produce a Happy grammar. We therefore believe that such custom preprocessors would better be implemented in a framework like SugarHaskell that supports the composition of many language extensions to be used in the same source file.

Priebe [2005] proposes a light-weight framework to implement preprocessors using Template Haskell. His key idea is use an universal preprocessor that wraps a Haskell source file in a call to a Template Haskell macro. The actual preprocessing is then done by the macro, which can be defined in a library. Unlike SugarHaskell, Priebe's approach does not address syntactic extensions or the composition of different preprocessing libraries. Nevertheless, the idea of combining a preprocessor (to define concrete syntax) and Template Haskell (to define desugarings) seems promising. Future work could investigate whether and how such a combined approach can be implemented as a SugarHaskell library.

The Utrecht Haskell compiler (UHC) [Dijkstra et al. 2009] is an extensible compiler for Haskell. It is heavily based on preprocessors that compose implementation fragments for different language levels. Extensions have to be compiled into UHC. Parsing is implemented with a hand-written combinator parser. In contrast, SugarHaskell supports extensions as libraries and declarative grammar extensions.

6.3 Camlp5

Camlp5[16] (as well as Camlp4) is an extensible preprocessor and pretty-printer that is targeted especially at extending the syntax of the functional language OCaml. Similar to SugarHaskell, Camlp5 parses a source file and applies desugaring rules. The resulting OCaml abstract syntax tree is either directly handed over to an OCaml compiler or pretty-printed into a text file.

The parsing stage of Camlp5 is based on an in-memory representation of the current grammar that is interpreted by a recursive descent parser. Backtracking can be enabled or disabled for individual rules. The parser operates on token streams that are produced by a separate, hand-written scanner. This choice of technology requires authors of language extension to consider operational details of the parser framework in order to correctly specify syntax extensions. Syntax extensions are written in OCaml and mutate the current grammar in order to install additional rules. The parsing stage of SugarHaskell is based on scannerless generalized parsing and therefore allows more declarative specification of grammars.

The desugaring stage of Camlp5 is based on desugaring rules written in OCaml. Desugaring rules have to fully desugar a language extension into OCaml abstract syntax trees. In SugarHaskell, desugaring rules are written in Stratego and they can produce code written with arbitrary language extensions, because the result of rule application will be further desugared, if necessary.

Like SugarHaskell, Camlp5 is self-applicable and in fact, Camlp5 includes several extensions that are targeted at language extension authors, including a special syntax for grammar rules and desugarings and support for concrete syntax in transformations. The successful use of self-application in Camlp5 shows how important that feature is.

Camlp5 can also be adapted for other languages than OCaml. However, language extensions themselves would still have to be written in OCaml, so the resulting system would not be as self-applicable as Camlp5 for OCaml is. We also believe that the more declarative approach of SDF and Stratego fits better with Haskell than the extension-by-mutation philosophy of Camlp5.

7. Conclusion

Syntactic concerns are important for programmers in practice. While semantics make code run, it is syntax that programmers interact with every day. Therefore, we believe it is important to support programmers in describing not only what their programs do, but also how their programs look. SugarHaskell addresses this belief and provides programmers with syntactic extensibility that allows extensions to use the full class of context-free languages enriched with layout sensitivity. SugarHaskell extensions compose and can affect object language and metalanguages equally easily. While there are some open issues regarding integration with Cabal, HackageDB, and Haskell's type system, SugarHaskell is operational and we invite programmers and language designers to experiment with SugarHaskell and its IDE.

Acknowledgments

We would like to thank Yi Dai, Stefan Fehrenbach, Paolo G. Giarrusso, and the anonymous reviewers for valuable feedback on earlier versions of this work. This work is supported in part by the European Research Council, grant No. 203099.

References

G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. Available at `http://bracha.org/pluggableTypesPosition.pdf`.

M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 157–172. Springer, 2005.

W. Clinger and J. Rees. Macros that work. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.

A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of Haskell Symposium*, pages 93–104. ACM, 2009.

T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In

[12] `http://www.cse.unsw.edu.au/~chak/haskell/c2hs/`

[13] `http://hackage.haskell.org/package/greencard`

[14] `http://projects.haskell.org/cpphs/`

[15] `http://personal.cis.strath.ac.uk/conor.mcbride/pub/she`

[16] `http://pauillac.inria.fr/~ddr/camlp5/`

Proceedings of Conference on Generative Programming and Component Engineering (GPCE), pages 167–176. ACM, 2011a.

S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011b.

S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012a. to appear.

S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. Submitted to *Conference on Software Language Engineering (SLE)*, June 2012b. Draft available at `http://sugarj.org/layout-parsing.pdf`.

GHC Team. The glorious Glasgow Haskell Compilation System user's guide, version 7.4.1, 2012.

B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 3–13. ACM, 2003.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: A case study in transformation modularity. *Software and System Modeling*, 9(3):375–402, 2010.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

P. J. Layzell. The history of macro processors in programming language extensibility. *The Computer Journal*, 28(1):29–33, 1985.

D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of Haskell Workshop*, pages 73–82. ACM, 2007.

S. Marlow (editor). Haskell 2010 language report. Available at `http://www.haskell.org/onlinereport/haskell2010`, 2010.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

M. D. McIlroy. Macro instruction extensions of compiler languages. *Communication of the ACM*, 3(4):214–220, 1960.

N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

R. Paterson. A new notation for arrows. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 229–240. ACM, 2001.

R. Paterson and S. Peyton Jones. Type and translation rules for arrow notation in GHC, 2004.

S. Priebe. Preprocessing Eden with Template Haskell. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 357–372. Springer, 2005.

F. Rieger. A language-independent framework for syntactic extensibility. Bachelor's Thesis, University of Marburg, June 2012. Submitted.

T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of Haskell Workshop*, pages 1–16. ACM, 2002.

M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2): 39–54, 2010.

E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 2051 of *LNCS*, pages 357–362. Springer, 2001.

E. Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. `http://voelter.de/data/pub/VoelterSolomatov_SLE2010_LanguageModularizationAndCompositionLWBs.pdf`, 2010.