Layout-sensitive Generalized Parsing

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann

University of Marburg, Germany

Abstract. The theory of context-free languages is well-understood and context-free parsers can be used as off-the-shelf tools in practice. In particular, to use a context-free parser framework, a user does not need to understand its internals but can specify a language *declaratively* as a grammar. However, many languages in practice are not context-free. One particularly important class of such languages is layout-sensitive languages, in which the structure of code depends on indentation and whitespace. For example, Python, Haskell, F#, and Markdown use indentation instead of curly braces to determine the block structure of code. Their parsers (and lexers) are not declaratively specified but hand-tuned to account for layout-sensitivity.
To support *declarative* specifications of layout-sensitive languages, we propose a parsing framework in which a user can annotate layout in a grammar. Annotations take the form of constraints on the relative positioning of tokens in the parsed subtrees. For example, a user can declare that a block consists of statements that all start on the same column. We have integrated layout constraints into SDF and implemented a layout-sensitive generalized parser as an extension of generalized LR parsing. We evaluate the correctness and performance of our parser by parsing 33 290 open-source Haskell files. Layout-sensitive generalized parsing is easy to use, and its performance overhead compared to layout-insensitive parsing is small enough for practical application.

1 Introduction

Most computer languages prescribe a textual syntax. A parser translates from such textual representation into a structured one and constitutes the first step in processing a document. Due to the development of parser frameworks such as lex/yacc [14], ANTLR [17,16], PEGs [5,6], parsec [12], or SDF [7], parsers can be considered off-the-shelf tools nowadays: Non-experts can use parsers, because language specifications are declarative. Although many parser frameworks support some form of context-sensitive parsing (such as via semantic predicates in ANTLR [17]), one particularly relevant class of languages is not supported declaratively by any existing parser framework: layout-sensitive languages.

Layout-sensitive languages were first proposed by Landin in 1966 [11]. In layout-sensitive languages, the translation from a textual representation to a structural one depends on the code's layout and its indentation. Most prominently, the *offside rule* prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure. In other

```
if x != y:
   if x > 0:
      y = x
else:
   y = -x
```

(a) Python: Indentation resolves the dangling else problem.

```
do input <- readInput
   case input of
     Just txt -> do putStrLn "thank you"
                    sendToServer txt
                    return True
     Nothing  -> fail "no input"
```

(b) Haskell: Nested block structure.

Fig. 1. Layout-sensitive languages use indentation instead of curly braces.

words, a token is offside if it occurs further to the left than the starting token of a structure; an offside token denotes the start of the next structure. In languages that employ the offside rule, the block structure of code is determined by indentation and layout alone, whose use is considered good style anyway.

The offside rule has been applied in a number of computer languages including Python, Haskell, F#, and Markdown. The Wikipedia page for the off-side rule[1] lists 20 different languages that apply the offside rule. For illustration, Figure 1 shows a Python and a Haskell program that use layout to declare the code's block structure. The layout of the Python program specifies that the **else** branch belongs to the outer **if** statement. Similarly, the layout of the Haskell program specifies to which **do**-block each statement belongs. Unfortunately, current declarative parser frameworks do not support layout-sensitive languages such as Python or Haskell, which means that often the manually crafted parsers in compilers are the only working parsers. This makes it unnecessarily hard to create tools for these language, such as refactoring tools or IDEs.

Our core idea is to declare layout as constraints on the shape and relative positioning of syntax trees. These *layout constraints* occur as annotations of productions in the grammar and restrict the applicability of annotated productions to valid layout. For example, for conditional expressions in Python, we annotate (among other things) that the **if** keyword must start on the same column as the **else** keyword and that all statements of a then or **else** branch must be further indented than the **if** keyword. These latter requirements are context-sensitive, because statements are rejected based on their appearance within a conditional statement. Thus, layout constraints cannot be fully enforced during the execution of a context-free parser.

We developed an extension of SDF [7] that supports layout constraints. The standard parsing algorithm for SDF is scannerless generalized LR parsing [20]. In a generalized parsing algorithm, all possible parse trees for an input string are processed in parallel. One approach to supporting layout would be to parse the input irrespective of layout in a first step (generating every possible parse tree), and then in a second step discard all syntax trees that violate layout

[1] `http://en.wikipedia.org/w/index.php?title=Off-side_rule&oldid=496815186`

constraints. However, we found that this approach is not efficient enough for practical applications: For many programs, the parser fails to terminate within 30 seconds. To improve performance, we identified a subset of layout constraints that in fact does not rely on context-sensitive information and therefore can be enforced at parse time. We found that enforcing these constraints at parse time and the remaining constraints at disambiguation time is sufficiently efficient.

To validate the correctness and to evaluate the performance of our layout-sensitive parser, we have build layout-sensitive SDF grammars for Python and Haskell. In particular, we applied our Haskell parser to all 33 290 Haskell files in the open-source repository Hackage. We compare the result of applying our parser to applying a traditional generalized parser to the same Haskell files where block structure has been made explicit through curly braces. Our estudy empirically validates the correctness of our parser and shows that our layout-sensitive parser can compete with parsers that requires explicit block structure.

We make the following contributions:

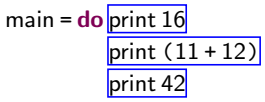
- We identify common idioms in existing layout-sensitive languages. Based on these idioms, we design a constraint language for specifying layout-sensitive languages declaratively.
- We identify context-free layout constraints that can be enforced at parse time to avoid excessive ambiguities.
- We implement a parser for layout-sensitive languages based on an existing scannerless generalized LR parser implementation in Java.
- We extended existing layout-insensitive SDF grammars for Python and Haskell with layout constraints.
- We evaluate the correctness and performance of our parser by parsing 33 290 open-source Haskell files and comparing the results against parse trees produced for Haskell files with explicit block structure. Our evaluation suggests that our parser is correct and fast enough for practical applications.

Our parser, grammars, and raw evaluation data are open-source and available online at `http://github.com/seba--/layout-parsing`. While our parser implementation is based on a scannerless parser, the ideas presented in this paper are applicable to parsers with separate lexers as well.

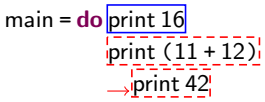
2 Layout in the wild

Many syntactic constructs in the programming language Haskell use layout to encode program structure. For example, the **do**-Block in the simple Haskell program in Figure 2(a) contains three statements which are vertically aligned at the same column in the source code. We visualize the alignment by enclosing the tokens that belong to a statement in a box. More generally, a box encloses code corresponding to a subtree of the parse tree. The exact meaning of these boxes will become clear in the next section, where they form the basis of our constraint language.

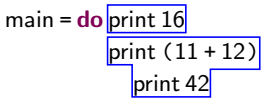
A Haskell parser needs to check the alignment of statements to produce correct parse trees. For example, Figure 2(b) visualizes an incorrect parse tree that



(a) Three statements with correct vertical alignment.



(b) Wrong parse: Statements have to begin in the same column, hence print 42 cannot be a statement.



(c) Correct parse: Only two statements, where the second print is applied to three arguments.

Fig. 2. Simple Haskell programs.

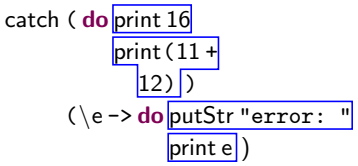
wrongly identifies print 42 as a separate statement, even though it is further indented than the other statements. Figure 2(c) visualizes the correct parse tree for this example: A **do**-Block with two statements. The second statement spans two lines and is parsed as an application of the function print to three arguments. In order to recognize program structure correctly, a parser for a layout-sensitive language like Haskell needs to distinguish programs as in Figure 2(a) from programs as in Figure 2(c).

It is not possible to encode this difference in a context-free grammar, because that would require counting the number of whitespace characters in addition to keeping track of nesting. Instead, many parsers for layout-sensitive languages contain a handwritten component that keeps track of layout and informs a standard parser for context-free languages about relevant aspects of layout, for instance by inserting special tokens into the token stream. For example, the Python language specification[2] describes an algorithm that preprocesses the token stream to delete some *newline* tokens and insert *indent* and *dedent* tokens when the indentation level changes. Python's context-free grammar assumes that this preprocessing step has already been performed, and uses the additional tokens to recognize layout-sensitive program structure.

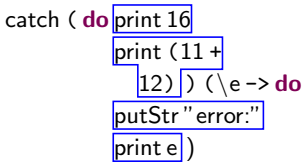
This approach has the advantage that a standard parser for context-free languages can be used to parse the preprocessed token stream, but it has the disadvantage that the overall syntax of the programming language is not defined in a declarative, human-readable way. Instead, the syntax is only defined in terms of a somewhat obscure algorithm that explicitly manipulates token streams. This is in contrast to the success story of declarative grammar and parsing technology [10].

Furthermore, a simple algorithm for layout-handling that informs a standard parser for context-free languages is not even enough to parse Haskell. The Haskell language specification describes that a statement ends earlier than visible from the layout if this is the only way to continue parsing [13]. For example, the Haskell program in Figure 3(a) is valid, and the statement print (11 + 12) only includes one closing parenthesis, because the second closing parenthesis could not be consumed inside the statement. An algorithm for layout handling could

[2] `http://docs.python.org/py3k/reference/`



(a) Exception handler.



(b) Means the same as (a).

Fig. 3. Both variants of this more complicated Haskell program have valid layout.

not decide where to end the statement by counting whitespace characters only. Instead, additional information from the context-free parser is needed to decide that the statement needs to end because the next token cannot be consumed. As a second and more extreme example, consider the program in Figure 3(b) that has the same parse tree as the program in Figure 3(a). In particular, the statements belong to different **do**-blocks even though they line up vertically. These two programs can only be parsed correctly by close cooperation between the context-free part of the parser and the layout-sensitive part of the parser, which therefore have to be tightly integrated. This need for tight integration further complicates the picture with the low-level, algorithmic specifications of layout rules prevalent in existing language specifications and implementations.

We have focused our investigation of layout-sensitive languages on Haskell and Python, but we believe our box model is general enough to explain layout in other languages as well.

3 Declaring layout with constraints

Our goal is to provide a high-level, declarative language for specifying and implementing layout-sensitive parsers. In the previous section, we have discussed layout informally. We have visualized layout by boxes around the tokens that belong to a subtree in Figures 2 and 3. We propose (i) to express layout rules formally as constraints on the shape and relative positioning of boxes and (ii) to annotate productions in a grammar with these constraints. The idea of layout constraints is that a production is only applicable if the parsed text adheres to the annotated constraint.

For example, Figure 4 displays an excerpt from our grammar for Haskell that specifies the layout of Haskell **do**-blocks with implicit (layout-based) as well as explicit block structure. This is a standard SDF grammar except that some productions are annotated with layout constraints. For example, the nonterminal Impl stands for implicit-layout statements, that is, statements of the form ⌜□ (but not □ or ⌜□). The layout constraint layout("1.first.col < 1.left.col") formally expresses the required shape ⌜□ for subtree number 1.

We provide the full grammar of layout constraints in Figure 5. Layout constraints can refer to direct subtrees (including terminals) of the annotated production through numerical indexes.

```
context-free syntax
                Stm  ->  Impl  {layout("1.first.col < 1.left.col")}
               Impl  ->  Impls
          Impl Impls ->  Impls {cons("StmSeq"), layout("1.first.col == 2.first.col")}
                Stm  ->  Expls
     Stm ";" Expls  ->  Expls {cons("StmSeq")}
              Impls  ->  Stms  {cons("Stms")}
     "{" Expls "}"  ->  Stms  {cons("Stms"), ignore-layout}
       "do" Stms    ->  Exp   {cons("Do"), longest-match}
```

Fig. 4. Excerpt of our layout-sensitive Haskell grammar. Statements with implicit layout (Impl) have to follow the offside rule. Multiple statements have to align vertically. Statements with explicit layout (Expl) are not layout-sensitive.

$$
\begin{aligned}
\mathit{tree} &::= \mathit{number} \\
\mathit{tok} &::= \mathit{tree}.\mathsf{first} \mid \mathit{tree}.\mathsf{left} \mid \mathit{tree}.\mathsf{right} \mid \mathit{tree}.\mathsf{last} \\
\mathit{ne} &::= \mathit{tok}.\mathsf{line} \mid \mathit{tok}.\mathtt{col} \mid \mathit{ne} + \mathit{ne} \mid \mathit{ne} - \mathit{ne} \\
\mathit{be} &::= \mathit{ne} == \mathit{ne} \mid \mathit{ne} < \mathit{ne} \mid \mathit{ne} > \mathit{ne} \mid \mathit{be} \,\&\&\, \mathit{be} \mid \mathit{be} \mid\mid \mathit{be} \mid !\mathit{be} \\
c &::= \mathsf{layout}(\mathit{be}) \mid \mathsf{ignore\text{-}layout}
\end{aligned}
$$

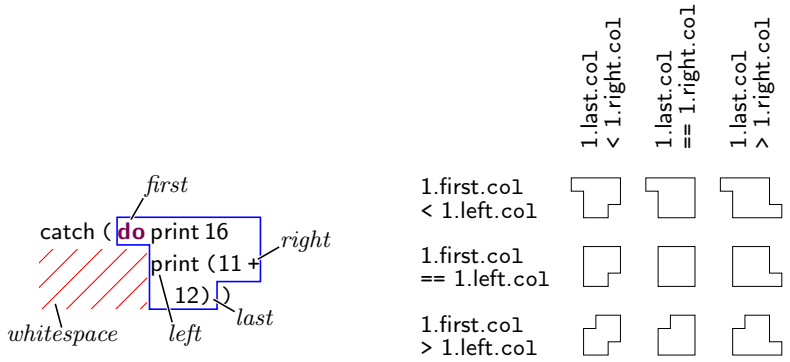
Fig. 5. Syntax of layout constraints c that can annotate SDF productions.

Each subtree exposes its shape via the source location of four tokens in the subtree, which describe the relevant positions in the token stream. Layout constraints use *token selectors* to access these tokens: first selects the first non-whitespace token, last selects the last non-whitespace token, left selects the leftmost non-whitespace token that is not on the same line as the first token, and right selects the rightmost non-whitespace token that is not on the same line as the last token. Figure 6(a) shows how the positions of these tokens describe the shape of a subtree. It is essential in our design that layout rules can be described in terms of the locations of these four tokens, because this provides a declarative abstraction over the exact shape of the source code. As is apparent from their definition, the token selectors left and right fail if all tokens occur in a single line. Since a single line of input satisfies any box shape, we do not consider this a constraint violation.

For each selected token, the *position selectors* line and col yield the token's line and column offset, respectively. Hence the constraint 1.first.col < 1.left.col specifies that the left border of the shape of subtree 1 must look like ⌜. In other words, the constraint 1.first.col < 1.left.col corresponds to Landin's offside rule. Consider the following example:

```
print (11 + 12)
  * 13
```

Here, 1.first selects the first token of the function name print, yielding the character p for scannerless parsers, or the token print otherwise. 1.left selects the left-most symbol not on the same line than print, that is, the operator symbol *.



(a) The source locations of four tokens induce (an abstraction of) the shape of a subtree.

(b) Layout constraints to restrict the shape of a box.

Fig. 6. Example layout constraints and the corresponding boxes.

This statement is valid according to the Impl production because the layout constraint is satisfied: The column in which print appears is to the left of the column in which * appears. Conversely, the following statement does not adhere to the shape requirement of Impl because the layout constraint fails:

```
print (11 + 12)
* 13
```

Consequently, the Impl production is not applicable to this statement.

The layout constraint 1.first.col < 1.left.col mentions only a single subtree of the annotated production and therefore restricts the shape of that subtree. Figure 6(b) shows other examples for layout constraints that restrict the shape of a subtree. In addition to these shapes, layout constraints can also prescribe the vertical structure of a subtree. For example, the constraint 1.first.line == 1.last.line prohibits line breaks within the subtree 1 and 1.first.line + num(2) == 1.last.line requires exactly two line breaks.

If a layout constraint mentions multiple subtrees of the annotated production, it specifies the relative positioning of these subtrees. For example, the nonterminal Impls in Figure 4 stands for a list of statements that can be used with implicit layout. In such lists, all statements must start on the same column. This vertical alignment is specified by the layout constraint 1.first.col == 2.first.col. This constraint naturally composes with the constraint in the Impl production: A successful parse includes applications of both productions and hence checks both layout constraints.

The anti-constraint ignore-layout can be used to deactivate layout validation locally. In some languages such as Haskell and Python, this is necessary to support explicit-layout structures within implicit-layout structures. For example, the Haskell grammar in Figure 4 declares explicit-layout statement lists. Since

these lists use explicit layout {stmt;...;stmt}, no additional constraints are needed. However, Haskell allows code within an explicit-layout list to violate layout constraints imposed by surrounding constructs. Correspondingly, we annotate explicit-layout lists with ignore-layout, which enables us to parse the following valid Haskell program:

```
do print (11 + 12)
     print 13
   do { print 14 ;
print 15 }
   print 16
```

Our Haskell parser successfully parses this program even though the second statement seemingly violates the shape requirement on Impl. However, since the nested explicit statement list uses ignore-layout, we ignore all its tokens when applying the left or right token selector. Therefore, the left selector in the constraint of Impl fails to find a leftmost token that is not on the first line, and the constraint succeeds by default.

We deliberately kept the design of our layout-constraint language simple to avoid distraction. For example, we left out language support for abstracting over repeating patterns in layout constraints. However, such facilities can easily be added on top of our core language. Instead, we focus on the integration of layout constraints into generalized parsing.

4 Layout-sensitive parsing with SGLR

We implemented a layout-sensitive parser based on our extension of SDF [7] with layout constraints. Our parser implementation builds on an existing Java implementation [9] of scannerless generalized LR (SGLR) parsing [18,20]. A SGLR parser processes all possible interpretations of the input stream in parallel and produces multiple potential parse results. Invalid parse results can be filtered out in an additional disambiguation phase.

We have modified the SGLR parser to take layout constraints into account.[3] As a first naive but correct strategy, we defer all validation of layout constraints until disambiguation time. As an optimization of this strategy, we then identify layout constraints that can be safely checked at parse time.

4.1 Disambiguation-time rejection of invalid layout

SDF distinguishes two execution phases: parse time and disambiguation time. At parse time, the SGLR parser processes the input stream to construct a *parse forest* of multiple potential parser results. This parse forest is input to the disambiguation phase, where additional information (e.g., precedence information) specified together with the context-free grammar is used to discard as many of

[3] We can reuse the parse-table generator without modifications, because it automatically forwards layout constraints from the grammar to the corresponding reduce-actions in the parse table.

the trees in the parse forest as possible. Ideally, only a single tree remains, which means that the given SDF grammar is unambiguous for the given input.

While conceptually layout constraints restrict the applicability of annotated productions, we can nevertheless defer the validation of layout constraints to disambiguation time. Accordingly, we first parse the input ignoring layout constraints and produce all possible trees. However, to enable later checking of token positions, during parsing we store line and column offsets in parse trees.

After parsing, we disambiguate the resulting parse forest by traversing it. Whenever we encounter the application of a layout-constrained production, we check that the layout constraint is satisfied. For violated constraints, we reject the corresponding subtree that used the production. If a layout violation occurs within an ambiguity node, we select the alternative result (if it is layout-correct).

The approach described so far is a generic technique that can be used to integrate any context-sensitive validation into context-free parsing. For instance, Bravenboer et al. [1] integrate type checking into generalized parsing to disambiguate metaprograms. However, layout-sensitive parsing is particularly hard because of the large number of ambiguities even in small programs.

For example, in the following Haskell programs, the number of ambiguities grows exponentially with the number of statements:

```
foo = do print 1
```

```
foo = do print 1
          print 2
```

```
foo = do print 1
          print 2
          print 3
```

For the first program, the context-free parser results in a parse forest with one ambiguity node that distinguishes whether the number `1` is a separate statement or an argument to `print`. The second example already results in a parse forest with 7 ambiguity nodes; the third example has 31 ambiguity nodes. The number of ambiguities roughly quadruples with each additional statement.

Despite sharing between ambiguous parse trees, disambiguation-time layout validation can handle programs of limited size only. For example, consider the Haskell program that contains 30 repetitions of the statement `print 1 2 3 4 5 6 7 8 9`. After parsing, the number of layout-related ambiguities in this program is so big that it takes more than 20 seconds to disambiguate it. A more scalable solution to layout-sensitive parsing is necessary.

4.2 Parse-time rejection of invalid layout

The main scalability problem in layout validation is that ambiguities are not local. Without explicit block structure, it is not clear how to confine layout-based ambiguities to a single statement, a single function declaration, or a single class declaration. For example, in the `print` examples from the previous subsection, the numbers on the last line can be arguments to the `print` function on the first line. Similarly, when using indentation to define the span of if-then-else branches as in Python, every statement following the if-then-else can be either within the else branch or not. It would be good to restrict the extent of ambiguities to more fine-grained regions at parse time to avoid excessive ambiguities.

Internally, SGLR represents intermediate parser results as states in a graph-structured stack [18]. Each state describes (i) a region in the input stream, (ii) a nonterminal that can generate this input, and (iii) a list of links to the states of subtrees. When parsing can continue in different ways from a single state, the parser splits the state and follows all alternatives. For efficiency, SGLR uses local ambiguity packing [18] to later join such states if they describe the same region of the input and the same nonterminal (the links to subtrees may differ). For instance, in the ambiguous input `print (1 + 2 + 3)`, the arithmetic expression is described by a single state that corresponds to both `(1+2)+3` and `1+(2+3)`. Thus, the parser can ignore the local ambiguity while parsing the remainder of the input.

Due to this sharing, we cannot check context-sensitive constraints at parse time. Such checks would require us to analyze and possibly resplit parse states that were joined before: Two parse states that can be treated equally from a context-free perspective may behave differently with respect to a context-sensitive property. For example, the context-free parser joins the states of the following two parse trees representing different Haskell statement lists:

The left-hand parse tree represents a statement list with two statements. The right-hand parse tree represents a statement list with a single statement that spans two lines. This statement violates the layout constraint from the Haskell grammar in Figure 4 because it does not adhere to the offside rule (shape ⌐). Since the context-free parser disregards layout constraints, it produces both statement lists nonetheless.

The two statement lists describe the same region in the input: They start and end at the same position. Moreover, both parse trees can be generated by the `Impls` nonterminal (Figure 4). Therefore, SGLR joins the parse states that correspond to the shown parse trees. This is a concrete example of two parse trees that differ with respect to a context-sensitive property, but are treated identically by SGLR.

Technically, context-sensitive properties require us to analyze and possibly split parse states that are not root in the graph-structured stack. Such a split deep in the stack would force us to duplicate all paths from root states to the split state. This not only entails a serious technical undertaking but likely degrades the parser's runtime and memory performance significantly.

To avoid these technical difficulties, we would like to enforce only those layout constraints at parse time that do not interact with sharing. Such constraints must satisfy the following invariant: If a constraint rejects a parse tree, it must also reject all parse trees that the parser might represent through the same parse state. For constraints that satisfy this invariant, it cannot happen that we prematurely reject a parse state that should have been split instead: All trees represented by that state would be rejected by the constraint. One class of layout constraints that adheres to this invariant is only using information that is encoded in the parse state itself, namely the input region and the nonterminal.

This information is the same for all represented trees and we can use it at parse time to reject states without influencing splitting or joining.

In our constraint language, the input region of a tree is described by the token selectors first and last. Since the input region is the same for all trees that share a parse state, constraints that only use the first and last token selectors (but not left or right) can be enforced at parse time without influencing sharing: If such a constraint rejects any random tree of a parse state, the constraint also rejects all other trees because they describe the same input region.

One particularly useful constraint that only requires the token selectors first and last is `1.first.col == 2.first.col`, which denotes that trees `1` and `2` need to be vertically aligned. Such constraint is needed for statement lists of both Haskell and Python. Effectively, the constraint reduces the number of potential statements to those that start on the same column. This confines many ambiguities to a single statement. For example, the constraint allows us to reject the program shown in Figure 2(b) because the statements are not aligned. However, it does not allow us to reject or distinguish the programs shown in Figure 2(a) and 2(c); we retain an ambiguity that we resolve at disambiguation time.

Technically, we enforce constraints at parse time when executing reduce actions. Specifically, in the function DO-REDUCTIONS [20], for each list of subtrees, we validate that the applied production permits the layout of the subtrees. If the production does not specify a layout constraint, the constraint is satisfied, or the constraint cannot be checked at parse time, the regular reduce action is performed. If a layout constraint is violated, the reduce action is skipped.

The remaining challenge is to validate that we in fact reduce ambiguity to a level that allows acceptable performance in practice.

5 Evaluation

We evaluate *correctness* and *performance* of our layout-sensitive generalized parsing approach with an implementation of a Haskell parser. Correctness is interesting because we reject potential parser results based on layout constraints; we expect that layout should not affect correctness. Performance is critical because our approach relies on storing additional position information and creating additional ambiguity nodes that are later resolved, which we expect to have a negative influence on performance. We want to assess whether the performance penalty of our approach is acceptable for practical use (e.g., in an IDE). Specifically, we evaluate the following research questions:

RQ1: Can a layout-sensitive generalized Haskell parser parse the same files and produce equivalent parse trees as a layout-insensitive Haskell parser that requires explicit layout?

RQ2: What is the performance penalty of the layout-sensitive Haskell parser compared to a layout-insensitive Haskell parser that requires explicit layout?

5.1 Research method

In a controlled setting, we quantitatively compare the results and performance of different Haskell parsers on a large set of representative Haskell files.

Parsers and parse results. We have implemented the layout-sensitive parser as discussed above by modifying the original SGLR parser written in Java.[4] We have extended an existing SDF grammar for Haskell that required explicit layout[5] with layout constraints. We want to compare our parser to a reimplementation of GHC's hand-tuned LALR(1) parser that has been developed by others and is deployed as part of the haskell-src-exts package.[6] Here, we refer to it simply as GHC parser. However, comparing the performance of our layout-sensitive SGLR parser to the hand-optimized GHC parser would be unfair since completely different parsing technologies are used. Also comparing the produced abstract syntax trees of both parsers is not trivial, because differently structured abstract syntax trees are generated. Therefore, we primarily compare our layout-sensitive parser to the original SGLR parser that did not support layout.

However, the original SGLR parser is layout-insensitive and therefore not able to parse Haskell files that use implicit layout (which almost all Haskell files do). Therefore, we also used the pretty printer of the haskell-src-exts package to translate Haskell files with arbitrary combinations of explicit and implicit layout into a representation with only explicit layout. Since the pretty printer also removes comments, the files may be smaller and hence faster to parse. Therefore, we use the same pretty printer to create a file that uses only implicit layout and contains no comments either.

Overall, we have three parsers (GHC, the original SGLR parser, and our layout-sensitive SGLR parser) which we can use to parse three different files (original layout, explicit-only layout, implicit-only layout). We are interested in the parser result and parse time of four combinations:

`GHC`. Parsing the file with *original layout* using the GHC parser.
`SGLR-Orig`. Parsing the file with *original layout* (possible mixture of explicit and implicit layout) with our layout-sensitive SGLR parser.
`SGLR-Expl`. Parsing the file after pretty printing with *explicit layout only* and without comments with the original SGLR parser.
`SGLR-Impl`. Parsing the file after pretty printing with *implicit layout only* and without comments with our layout-sensitive SGLR parser.

We illustrate the process, the parsers, and the results in Figure 7. All SGLR-based parsers use the same Haskell grammar of which the original SGLR parser ignores the layout constraints. Our Haskell grammar implements the Haskell 2010 language report [13], but additionally supports the following extensions to

[4] Actually, we improved the original implementation by eliminating recursion to avoid stack overflows when parsing files with long comments or long literal strings.
[5] `http://strategoxt.org/Stratego/HSX`
[6] `http://hackage.haskell.org/package/haskell-src-exts`

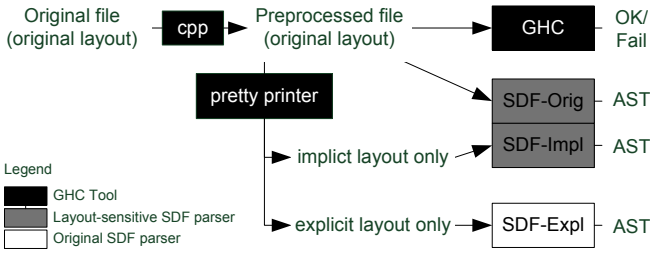


Fig. 7. Evaluation setup

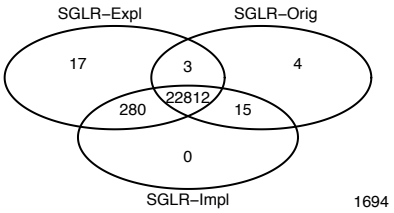
increase coverage of supported files: *HierarchicalModules*, *MagicHash*, *FlexibleInstances*, *FlexibleContexts*, *GeneralizedNewtypeDeriving*. We configured the GHC parser accordingly and, in addition, deactivated its precedence resolution of infix operators, which is a context-sensitive mechanism that can be implemented as a post-processing step. Running the C preprocessor is necessary in many files and performed in all cases. Note that `SGLR-Orig` and `SGLR-Impl` use the same parser, but execute it on different files.

Subjects. To evaluate performance and correctness on realistic files, we selected a large representative collection of Haskell files. We attempt to parse all Haskell files collected in the open-source Haskell repository Hackage.[7] We extracted the latest version of all 3081 packages that contain Haskell source code on May 15, 2012. In total, these packages contain 33 290 Haskell files that amount to 258 megabytes and 5 773 273 lines of Haskell code (original layout after running cpp).

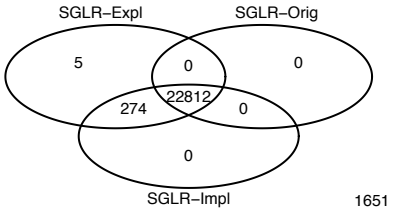
Data collection. We perform measurements by repeating the following for each file in Hackage: We run the C preprocessor and the pretty printer to create the files with original, explicit-only, and implicit-only layout. We measure the wall-clock time of executing the GHC parser and the SGLR-based parsers on the prepared files as illustrated in Figure 7. We stop parsers after a timeout of 30 seconds and interpret longer parsing runs as failure. We parse all files in a single invocation of the Java virtual machine and invoke the garbage collector between each parser execution. After starting the virtual machine, we first parse 20 packages (215 files) and discard the results to account for warmup time of Java's JIT compiler. A whole run takes about 6 hours. We repeat the entire process with all measurements three times after system reboots and use the arithmetic mean of each file and parser over all runs.

We run all performance measurements on the same 3 GHz, dual-core machine with 4GB memory and Java Hotspot VM version 1.7.0_04. We specified a maximum heap size of 512MB and a maximum stack size of 16MB.

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



(a) Number of files each parser produces the correct AST for.



(b) Ignoring files that timeout with at least one parser.

Fig. 8. Correctness of layout-sensitive parsing.

Analysis procedure. We discard all files that cannot be parsed by the GHC parser configured as described above. On the remaining files, for research question RQ1 (correctness), we evaluate that the three abstract syntax trees produced by SGLR parsers are the same (that is, we perform a form of differential testing).

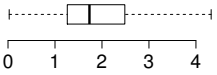
For research question RQ2 (performance penalty), we determine the relative slow down between `SGLR-Expl` and `SGLR-Impl` (and briefly compare also the performance of the other parsers). We calculate the relative performance penalty between parsers separately for each file that can be parsed by all three parsers. We report the geometric mean and the distribution of the relative performance of all these files.

5.2 Results

Correctness. Of all 33 290 files, 9071 files (27 percent) could not be parsed by the GHC parser (we suspect the high failure rate is due to the small number of activated language extensions). Of the remaining 24 219 files, 22 812 files (94 percent) files could be parsed correctly with all three SGLR-based parsers (resulting in the same abstract syntax tree). We show the remaining numbers in the Venn diagram in Figure 8(a). Some differences are due to timeouts; the diagram in Figure 8(b) shows those results that do not time out in any parser.

Performance. The median parse times per file of all parsers are given in Figure 9(b). Note that the results for `GHC` are not directly comparable, since they include a process invocation, which corresponds to an almost constant overhead of 15 ms. On average `SGLR-Impl` is 1.8 times slower than `SGLR-Expl`. We show the distribution of performance penalties as box plot in Figure 9(a) (without outliers). The difference between `SGLR-Orig` and `SGLR-Impl` is negligible; `SGLR-Impl` is slightly faster on average because pretty printing removes comments.

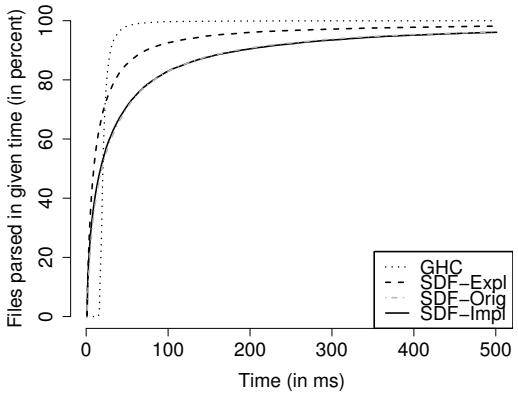
In Figure 9(c), we show the parse times for all four parsers (the graph shows how many percent of all files can be parsed within a given time). We see that, as to be expected, `SGLR-Expl` is slower than the hand-optimized `GHC`, and `SGLR-Impl` is slower than `SGLR-Expl`. The parsers `SGLR-Impl` and `SGLR-Orig` perform similarly and are essentially not distinguishable in this figure.



(a) Distribution of relative performance penalty (`SGLR-Impl`/`SGLR-Expl`).

	median time/file
`GHC`	(<)19 ms
`SGLR-Expl`	8 ms
`SGLR-Orig`	18 ms
`SGLR-Impl`	17 ms

(b) Median parse times.



(c) Distribution of parsing times.

Fig. 9. Performance of layout-sensitive parsing.

5.3 Interpretation and discussion

As shown in Figure 8(a), `SGLR-Orig` and `SGLR-Impl` do not always produce the same result as `SGLR-Expl`. Of these differences, 40 can be ascribed to timeouts, which occur in `SGLR-Expl` as well as in `SGLR-Orig` and `SGLR-Impl`. The remaining differences are shown in Figure 8(b). We investigated these differences and found that the five files that only `SGLR-Expl` can parse are due to Haskell statements that start with a pragma comment, for example:

```
{-# SCC "Channel_Write" #-} liftIO . atomically $ writeTChan pmc m
```

Since our SGLR-based parsers ignore such pragma comments, the statement appears to be indented too far. We did not further investigate due to the low number of occurrences of this pattern.

For the 274 files that only `SGLR-Expl` and `SGLR-Impl` can parse, we took samples and found that `SGLR-Orig` failed because of code that uses a GHC extension called *NondecreasingIndentation*, which is not part of the Haskell 2010 language report but cannot be deactivated in the GHC parser. The extension allows programs to violate the offside rule for nested layout blocks:

```
foo = do                          foo = do
  print 16                          print 16
  do            pretty-prints to    do
  print 17                            print 17
  print 18                            print 18
```

None of the SGLR-based parsers can handle such programs. However, the GHC pretty printer always produces code that does not require *NondecreasingIndentation*. Thus, `SGLR-Expl` and `SGLR-Impl` can parse the pretty-printed code, whereas `SGLR-Orig` fails on the original code. We consider this a bug of the

reimplementation of the GHC parser, which does not implement the Haskell 2010 language report even when configured accordingly.

Finally, GHC accepts 1651 files that none of the SGLR-based parsers accepts. Since not even the layout-insensitive parser `SGLR-Expl` accepts these files, we suspect inaccuracies in the original Haskell grammar that are independent of layout.

Regarding performance, layout-sensitive parsing with `SGLR-Impl` entails an average slowdown of 1.8 compared to layout-insensitive parsing with `SGLR-Expl`. Given the median parse times per file (Figure 9(b)), this slowdown is still in the realm of a few milliseconds and suggests that layout-sensitive parsing can be applied in practice. In particular, this slowdown seems acceptable given the benefits of declarative specifications of layout as in our approach, as opposed to low-level implementation of layout within a lexer or the parser itself. Furthermore, we expect room for improving the performance of our implementation of layout-sensitive parsing, as we discuss in Section 6.

Overall, regarding correctness (RQ1), we have shown that layout-sensitive parsing can parse almost all files that the layout-insensitive `SGLR-Expl` can parse. In fact, we did not find a single actual difference that would indicate an incorrect parse. Regarding performance penalty (RQ2), we believe that the given slowdown does not inhibit practical application of our parser.

5.4 Threats to validity

A key threat to external validity (generalizability of the results) is that we have analyzed only Haskell files and parse only files from the Hackage repository. We believe that the layout mechanisms of Haskell are representative for other languages, but our evaluation cannot generalize beyond Haskell. Furthermore, files in Hackage have a bias toward open-source libraries. However, we believe that our sample is large enough and the files in Hackage are diverse enough to present a general picture.

An important threat to internal validity (factors that allow alternative explanations) is the pretty printing necessary for parser `SGLR-Expl`. Pretty printing removes comments but possibly adds whitespace. The pretty-printed files with explicit layout have a 45 percent larger overall byte size compared to original layout, whereas the pretty-printed files with implicit layout have a 15 percent smaller byte size. Unfortunately, we have no direct influence on the pretty printer. We believe that the influence of pretty printing is largely negligible, because whitespace and comments should not trigger ambiguities during parsing (the similarity of the performance of `SGLR-Orig` and `SGLR-Impl` can be seen as support). However, a more configurable pretty printer should improve internal validity in future work.

It may be surprising that GHC (and also `SGLR-Orig`) fail to parse over one quarter of all files. We have sampled some of these files and found that they require more language extensions than we currently support. For example, the GADTs and TypeFamilies extensions seem to be popular, but we did not implement their syntax in our grammar and deactivated them in the GHC parser.

In future work, we would like to support Haskell more completely, which should increase the number of supported Hackage files.

Regarding construct validity (suitability of metrics for evaluation goal), we measured performance using wall-clock time only. For the SGLR-based parsers, we control JIT compilation with a warmup phase. By running the garbage collector between parser runs and monitoring the available memory, we ensured that all parsers have a similar amount of memory available. However, the layout-aware parser stores additional information and may perform different in scenarios with less memory available. Furthermore, we can, of course, not entirely eliminate background noise. Although we have repeated all measurements only three times, we believe the measurements are sufficiently clear and we have checked that variations between the three measurements are comparably minor for all parsers (for over 95 percent of all files, the standard deviation of these measurements was less than 10 percent of the mean).

6 Discussion and future work

We modified an SGLR parser to support validation of layout constraints at parse time and disambiguation time. Here, we summarize some technical implications, potential improvements, and limitations of our parser.

Technical implications. Layout-sensitive parsing interacts with traditional disambiguation methods such as priorities or follow restrictions. For example, consider the following Haskell program, which can be parsed into two layout-correct parse trees (boxes indicate the toplevel structure of the trees):

In both parse trees, the **do**-block consists of a single statement that adheres to the offside rule. However, the Haskell language report specifies that the left-hand parse tree is correct: For **do**-blocks the *longest match* needs to be selected.

SDF provides a longest-match disambiguation filter for lexical syntax, called follow restrictions [19]. A typical use of follow restrictions is to ensure that identifiers are not followed by any letters, which should be part of the identifier instead. Since, in fact, both of the above parse trees correspond to some valid Haskell program (dependent on layout), not even context-free follow restrictions enable us to disambiguate correctly because they ignore layout. Similarly, a priority filter would reject the same parse tree irrespective of layout.

For this reason, we added a disambiguation filter to SDF called `longest-match`. We use it to declare that, in case of ambiguity, a production should extend as far to the right as possible. We annotated the production for **do**-blocks in Figure 4 accordingly. Since our parser stores position information in parse trees anyway, the implementation of the longest-match filtering is simple: For ambiguous applications of a longest-match production we compare the position of the `last` tokens and choose the tree that extends further.

More generally, it should be noted that due to position information in parse trees, our parser supports less sharing than traditional GLR parsers do. Essentially, our parser can only share parse trees that describe the same region in the input stream. We have not yet investigated the implications on memory consumption, but our empirical study indicates that the performance penalty is acceptable.

Performance improvements. In our implementation of layout-sensitive generalized parsing, we mostly focused on correctness and only addressed performance in so far as it influences the feasibility of our approach. Therefore, in our current implementation, we suspect two significant performance improvements are still possible. First, we *interpret* layout constraints by recursive-descent with dynamic type checking. We have profiled the performance of our parser and found that about 25 percent of parse time and disambiguation time are spent on interpreting layout constraints. We expect that a significant improvement is possible by *compiling* layout constraints when loading the parse table. Second, our current implementation validates *all* layout constraints at disambiguation time. However, we validate many constraints at parse time already (as described in Section 4.2). We suspect that avoiding the repeated evaluation of those constraints represents another significant performance improvement.

Limitations. In general, context-sensitive properties can be validated after parsing at disambiguation time without restriction. However, the expressivity of our constraint language is limited in multiple ways. First, layout constraints in our language are *compositional*, that is, a constraint can only refer to the direct subtrees of a production. It might be useful to extend our constraint language with pattern-matching facilities as known from XPath [21]. However, it is not obvious how such pattern matching influences the performance of parsing and disambiguation; we leave this question open. A second limitation is that we focus on one-dimensional layout-sensitive languages only. However, a few layout-sensitive languages employ a two-dimensional syntax, for example, for type rules as in Epigram [15]. We would like to investigate whether our approach to layout-sensitivity generalizes to two-dimensional parsers.

7 Related work

We have significantly extended SDF's frontend [7] and its SGLR backend [18,20] to support layout-sensitive languages declaratively. We are not aware of any other parser framework that provides a *declarative* mechanism for layout-sensitive languages. Instead, existing implementations of parsers for layout-sensitive languages are handwritten and require separate layout-sensitive lexing.

For example, the standard Python lexer and parser are handwritten C programs.[8] While parsing, the lexer checks for changes of the indentation level in the input, and marks them with special *indent* and *dedent* tokens. The parser

[8] `http://svn.python.org/projects/python/trunk/Modules/parsermodule.c`

then consumes these tokens to process layout-sensitive program structures. This implementation is non-declarative.

As another example, the GHC Haskell compiler employs a layout-sensitive lexer that uses the Lexer generator Alex[9] in combination with manual Haskell code. The generated layout-sensitive lexer manages a stack of layout contexts that stores the beginning of each layout block. When the parser queries the lexer for layout-relevant tokens (such as curly braces), the lexer adapts the layout context accordingly. These interactions between parser and lexer are non-trivial and require virtual tokens for implicit layout. Since the layout rules of Haskell are hard-coded into the lexer, it is also not easy to adapt the parser and lexer for other languages. The same holds for the Utrecht Haskell Compiler [2].

Data-dependent grammars [8] support the declaration of constraints to restrict the applicability of a production. However, constraints in data-dependent grammars must be context-insensitive [8, Lemma 4], and therefore cannot be used to describe languages with context-sensitive layout such as Haskell.

8 Conclusion

We have presented a parser framework that allows the declaration of layout constraints within a context-free grammar. Our generalized parser enforces constraints at parse time when possible but fully validates parse trees at disambiguation time. We have empirically shown that our parser is correct and the performance penalty is acceptable compared to layout-insensitive generalized parsing. We believe that this work will enable language implementors to specify the grammar of their layout-sensitive languages in a high-level, declarative way.

Our original motivation for this work was to develop a syntactically extensible variant of Haskell in the style of SugarJ [3], where regular programmers write syntactic language extensions. This requires a declarative and extensible syntax formalism as provided by SDF. Based on the work presented here, we have been able to implement SugarHaskell [4], an extensible preprocessor and IDE for Haskell.

Acknowledgments. We thank Doaitse Swierstra for discussion and challenging Haskell examples, and the anonymous reviewers for their feedback. This work is supported in part by the European Research Council, grant No. 203099.

References

1. 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.
2. 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.

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

3. 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, 2011.
4. S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*. ACM, 2012. to appear.
5. B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 36–47. ACM, 2002.
6. B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 111–122. ACM, 2004.
7. 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.
8. T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 417–430. ACM, 2010.
9. L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments: Adding error recovery to scannerless generalized-LR parsing. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 445–464. ACM, 2009.
10. L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 918–932. ACM, 2010.
11. P. J. Landin. The next 700 programming languages. *Communication of the ACM*, 9(3):157–166, 1966.
12. D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.
13. S. Marlow (editor). Haskell 2010 language report. Available at `http://www.haskell.org/onlinereport/haskell2010`, 2010.
14. T. Mason and D. Brown. *Lex & yacc*. O'Reilly, 1990.
15. C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.
16. T. Parr and K. Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 425–436. ACM, 2011.
17. T. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.
18. M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.
19. M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 143–158. Springer, 2002.
20. E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.
21. XSL Working Group and XML Linking Working Group. XML Path language (XPath) 1.0. Available at `http://www.w3.org/TR/xpath/`, 1999.